The Atari BASIC SOURCE BOOK

A complete explanation of the inside workings of Atari BASIC, along with the original source code. For intermediate and advanced programmers.

Bill Wilkinson
Kathleen O'Brien
Paul Laughton
The Atari® BASIC

SOURCE BOOK

Compiled by Bill Wilkinson
Optimized Systems Software, Inc.

With the assistance of
Kathleen O’Brien and Paul Laughton
# Table of Contents

Publisher's Foreword ........................................... v
Acknowledgments ................................................... vii
Preface .......................................................... ix

## Part One: Inside Atari BASIC

1. Atari BASIC: A High-level Language Translator ........... 1
2. Internal Design Overview ................................... 7
3. Memory Usage ............................................... 13
4. Program Editor ............................................. 25
5. The Pre-compiler .......................................... 33
6. Execution Overview ........................................ 49
7. Execute Expression ........................................ 55
8. Execution Boundary Conditions ............................ 71
9. Program Flow Control Statements .......................... 75
10. Tokenized Program Save and Load ......................... 81
11. The LIST and ENTER Statements .......................... 85
12. Atari Hardware Control Statements ....................... 91
13. External Data I/O Statements ............................. 95
14. Internal I/O Statements .................................. 103
15. Miscellaneous Statements ................................ 105
16. Initialization .............................................. 109

## Part Two: Directly Accessing Atari BASIC

Introduction to Part Two ....................................... 113
1. Hexadecimal Numbers ...................................... 115
2. PEEKing and POKEing ...................................... 119
3. Listing Variables in Use .................................. 123
4. Variable Values ........................................... 125
5. Examining the Statement Table ............................ 129
6. Viewing the Runtime Stack ................................ 133
7. Fixed Tokens .............................................. 135
8. What Takes Precedence? .................................. 137
9. Using What We Know ...................................... 139

## Part Three: Atari BASIC Source Code

Source Code Listing ............................................. 143
Appendices

A  Macros in Source Code ...................................... 273
B  The Bugs in Atari BASIC ..................................... 275
C  Labels and Hexadecimal Addresses ......................... 281

Index ................................................................. 285
It's easy to take a computer language like Atari BASIC for granted. But every PEEK and POKE, every FOR-NEXT loop and IF-THEN branch, is really a miniprogram in itself. Taken together, they become a powerful tool kit. And, as Atari owners know, there are few home-computer languages as powerful and versatile — from editing to execution — as Atari BASIC.

With this book, the Atari BASIC tool kit is unlocked. The creators of Atari BASIC and COMPUTE! Publications now offer you, for the first time, a detailed, inside look at exactly how a major computer manufacturer's primary language works.

For intermediate programmers, the thorough and careful explanations in Parts 1 and 2 will help you understand exactly what is happening in your Atari computer as you edit and run your programs.

For advanced programmers, Part 3 provides a complete listing of the source code for Atari BASIC, so that your machine language programs can make use of the powerful routines built into that 8K cartridge.

And for programmers at all levels, by the time you're through studying this book you'll feel that you've seen a whole computer language at work.

Special thanks are due to Bill Wilkinson, the creative force behind Atari BASIC and many other excellent programs for Atari and other computers, for his willingness to share copyrighted materials with computer users. Readers of COMPUTE! Magazine already know him as a regular columnist, and in this book he continues his tradition of clear explanations and understandable writing.
Acknowledgments

As far as we know, this is the first time that the actual source listing of a major manufacturer’s primary computer language has been made available to the general public.

As with our previous COMPUTE! Publications book *Inside Atari DOS*, this book contains much more than simply a source listing. All major routines are examined and explained. We hope that when you finish reading this book you will have a better understanding of and appreciation for the design and work which go into as sophisticated a program as Atari BASIC.

This book is the result of the efforts of many people. The initial credit must go to Richard Mansfield of COMPUTE! Publications for serving as our goad and go-between. Without his (and COMPUTE!’s) insistence, this book might never have been written. Without his patience and guidance, the contents of this book might not have been nearly as interesting.

To Kathleen O’Brien and Paul Laughton must go the lion’s share of the authoring credits. Between them, they have done what I believe is a very creditable job of explaining a very difficult subject, the internal workings of Atari BASIC. In fact, Part I of this book is entirely their work. Of course, their ability to explain the listing may not be so surprising. After all, between them they wrote almost all of the original code for Atari BASIC. So, even though Paul and Kathleen are not associated with Optimized Systems Software, we were pleased to have their invaluable help in writing this book and hope that they receive some of the credit which has long been due them.

Mike Peters was responsible for taking our old, almost unreadable copies of the source code diskettes for Atari BASIC and converting them to another machine, using another assembler, and formatting the whole thing into an acceptable form for this book. This isn’t surprising either, since Mike keypunched the original (yes, on cards).

And I am Bill Wilkinson, the one responsible for the rest of this book. In particular, I hope you will find that a good amount of the material in Part II will aid you in understanding how to make the best use of this book.
The listing of Atari BASIC is reproduced here courtesy of OSS, Inc., which now owns its copyright and most other associated rights.
In 1978, Atari, Inc., purchased a copy of Microsoft BASIC for the 6502 microprocessor (similar to the version from which Applesoft is derived). After laboring for quite some time, the people of Atari still couldn’t make it do everything they wanted it to in the ROM space they had available. And there was a deadline fast approaching: the January 1979 Las Vegas Consumer Electronics Show (CES).

At that time, Kathleen, Paul, Mike and I all worked for Shepardson Microsystems, Inc. (SMI). Though little known by the public, SMI was reasonably successful in producing some very popular microcomputer software, including the original Apple DOS, Cromemco’s 16K and 32K BASICS, and more. So it wasn’t too surprising that Atari had heard of us.

And they asked us: Did we want to try to fix Microsoft BASIC for them? Well, not really. Did we think we could write an all-new BASIC in a reasonable length of time? Yes. And would we bet a thousand dollars a week on our ability to do so?

While Bob Shepardson negotiated with Atari and I wrote the preliminary specifications for the language (yes, I’m the culprit), time was passing all too rapidly. Finally, on 6 October 1978, Atari’s Engineering Department gave us the okay to proceed.

The schedule? Produce both a BASIC and a Disk File Manager (which became Atari DOS) in only six months. And, to make sure the pressure was intense, they gave us a $1000-a-week incentive (if we were early) or penalty (if we were late).

But Paul Laughton and Kathleen O’Brien plunged into it. And, although the two of them did by far the bulk of the work, there was a little help from Paul Krasno (who implemented the transcendental routines), Mike Peters (who did a lot of keypunching and operating), and me (who designed the floating point scheme and stood around in the way a lot). Even Bob Shepardson got into the act, modifying his venerable IMP-16 assembler to accept the special syntax table mnemonics that Paul invented (and which we paraphrase in the current listing via macros).
Atari delivered the final signed copy of the purchase order on 28 December 1978, two and a half months into the project. But it didn’t really matter: Paul and Kathy were on vacation, having delivered the working product more than a week before!

So Atari took Atari BASIC to CES, and Shepardson Microsystems faded out of the picture. As for the bonus for early delivery — there was a limit on how much the incentive could be. Darn.

The only really unfortunate part of all this was that Atari got the BASIC so early that they moved up their ROM production schedule and committed to a final product before we had a chance to do a second round of bug fixing.

And now? Mike and I are running Optimized Systems Software, Inc. And even though Paul and Kathleen went their own way, we have kept in touch enough to make this book possible.
Part One

How Atari BASIC Works
How Basic Works
The programming language which has become the de facto standard for the Atari Home Computer is the Atari 8K BASIC Cartridge, known simply as Atari BASIC. It was designed to serve the programming needs of both the computer novice and the experienced programmer who is interested in developing sophisticated applications programs. In order to meet such a wide range of programming needs, Atari BASIC was designed with some unique features.

In this chapter we will introduce the concepts of high level language translators and examine the design features of Atari BASIC that allow it to satisfy such a wide variety of needs.

Language Translators

Atari BASIC is what is known as a high level language translator. A language, as we ordinarily think of it, is a system for communication. Most languages are constructed around a set of symbols and a set of rules for combining those symbols.

The English language is a good example. The symbols are the words you see on this page. The rules that dictate how to combine these words are the patterns of English grammar. Without these patterns, communication would be very difficult, if not impossible: Out sentence this believe, of make don't this trying if sense you to! If we don't use the proper symbols, the results are also disastrous: @twu2 yeggopt gjsiem, keorw?

In order to use a computer, we must somehow communicate with it. The only language that our machine really understands is that strange but logical sequence of ones and zeros known as machine language. In the case of the Atari, this is known as 6502 machine language.

When the 6502 central processing unit (CPU) "sees" the sequence 01001000 in just the right place according to its rules of syntax, it knows that it should push the current contents of
the accumulator onto the CPU stack. (If you don’t know what an "accumulator" or a "CPU stack" is, don’t worry about it. For the discussion which follows, it is sufficient that you be aware of their existence.)

Language translators are created to make it simpler for humans to communicate with computers. There are very few 6502 programmers, even among the most expert of them, who would recognize 01001000 as the push-the-accumulator instruction. There are more 6502 programmers, but still not very many, who would recognize the hexadecimal form of 01001000, $48, as the push-the-accumulator instruction. However, most, if not all, 6502 programmers will recognize the symbol PHA as the instruction which will cause the 6502 to push the accumulator.

PHA, $48, and even 01001000, to some extent, are translations from the machine’s language into a language that humans can understand more easily. We would like to be able to communicate to the computer in symbols like PHA; but if the machine is to understand us, we need a language translator to translate these symbols into machine language.

The Debug Mode of Atari’s Editor/Assembler cartridge, for example, can be used to translate the symbols $48 and PHA to the ones and zeros that the machine understands. The debugger can also translate the machine’s ones and zeros to $48 and PHA. The assembler part of the Editor/Assembler cartridge can be used to translate entire groups of symbols like PHA to machine code.

**Assemblers**

An assembler — for example, the one contained in the Assembler/Editor cartridge — is a program which is used to translate symbols that a human can easily understand into the ones and zeros that the machine can understand. In order for the assembler to know what we want it to do, we must communicate with it by using a set of symbols arranged according to a set of rules. The assembler is a translator, and the language it understands is 6502 assembly language.

The purpose of 6502 assembly language is to aid program authors in writing machine language code. The designers of the 6502 assembly language created a set of symbols and rules that matches 6502 machine language as closely as possible.

This means that the assembler retains some of the
disadvantages of machine language. For instance, the process of adding two large numbers takes dozens of instructions in 6502 machine language. If human programmers had to code those dozens of instructions in the ones and zeros of machine language, there would be very few human programmers.

But the process of adding two large numbers in 6502 assembly language also takes dozens of instructions. The assembly language instructions are easier for a programmer to read and remember, but they still have a one-to-one correspondence with the dozens of machine language instructions. The programming is easier, but the process remains the same.

High Level Languages
High level languages, like Atari BASIC, Atari PILOT, and Atari Pascal, are simpler for people to use because they more closely approximate human speech and thought patterns. However, the computer still understands only machine language. So the high level languages, while seeming simple to their users, are really much more complex in their internal operations than assembly language.

Each high level language is designed to meet the specific need of some group of people. Atari Pascal is designed to implement the concept of structured programming. Atari PILOT is designed as a teaching tool. Atari BASIC is designed to serve both the needs of the novice who is just learning to program a computer and the needs of the expert programmer who is writing a sophisticated application program, but wants the program to be accessible to a large number of users.

Each of these languages uses a different set of symbols and symbol-combining rules. But all these language translators were themselves written in assembly language.

Language Translation Methods
There are two different methods of performing language translation — compilation and interpretation. Languages which translate via interpretation are called interpreters. Languages which translate via compilation are called compilers.

Interpreters examine the program source text and simulate the operations desired. Compilers translate the program source text into machine language for direct machine execution.
The compilation method tends to produce faster, more efficient programs than does the interpretation method. However, the interpretation method can make programming easier.

**Problems with the Compiler Method**

The compiler user first creates a program source file on a disk, using a text editing program. Then the compiler carefully examines the source program text and generates the machine language as required. Finally, the machine language code is loaded and executed. While this three-step process sounds fairly simple, it has several serious "gotchas."

Language translators are very particular about their symbols and symbol-combining rules. If a symbol is misspelled, if the wrong symbol is used, or if the symbol is not in exactly the right place, the language translator will reject it. Since a compiler examines the entire program in one gulp, one misplaced symbol can prevent the compiler from understanding any of the rest of the program — even though the rest of the program does not violate any rules! The result is that the user often has to make several trips between the text editor and the compiler before the compiler successfully generates a machine language program.

But this does not guarantee that the program will work. If the programmer is very good or very lucky, the program will execute perfectly the very first time. Usually, however, the user must debug the program.

This nearly always involves changing the source program, usually many times. Each change in the source program sends the user back to step one: after the text editor changes the program, the compiler still has to agree that the changes are valid, and then the machine code version must be tested again. This process can be repeated dozens of times if the program is very complex.

**Faster Programming or Faster Programs?**

The interpretation method of language translation avoids many of these problems. Instead of translating the source code into machine language during a separate compiling step, the interpreter does all the translation while the program is running. This means that whenever you want to test the program you’re writing, you merely have to tell the interpreter to run it. If things don’t work right, stop the program, make a few changes, and run the program again at once.
You must pay a few penalties for the convenience of using the interpreter’s interactive process, but you can generally develop a complex program much more quickly than the compiler user can.

However, an interpreter is similar to a compiler in that the source code fed to the interpreter must conform to the rules of the language. The difference between a compiler and an interpreter is that a compiler has to verify the symbols and symbol-combining rules only once — when the program is compiled. No evaluation goes on when the program is running. The interpreter, however, must verify the symbols and symbol-combining rules every time it attempts to run the program. If two identical programs are written, one for a compiler and one for an interpreter, the compiled program will generally execute at least ten to twenty times faster than the interpreted program.

**Pre-compiling Interpreter**

Atari BASIC has been incorrectly called an interpreter. It does have many of the advantages and features of an interpretive language translator, but it also has some of the useful features of a compiler. A more accurate term for Atari’s BASIC Language Translator is *pre-compiling interpreter*.

Atari BASIC, like an interpreter, has a text editor built into it. When the user enters a source line, though, the line is not stored in text form, but is translated into an intermediate code, a set of symbols called *tokens*. The program is stored by the editor in token form as each program line is entered. Syntax and symbol errors are weeded out at that time.

Then, when you run the program, these tokens are examined and their functions simulated; but because much of the evaluation has already been done, the execution of an Atari BASIC program is faster than that of a pure interpreter. Yet Atari BASIC’s program-building process is much simpler than that of a compiler.

Atari BASIC has advantages over compilers and interpreters alike. With Atari BASIC, every time you enter a line it is verified for language correctness. You don’t have to wait until compilation; you don’t even have to wait until a test run. When you type RUN you already know there are no syntax errors in your program.
Chapter Two

Internal Design Overview

Atari BASIC is divided into two major functional areas: the Program Constructor and the Program Executor. The Program Constructor is used when you enter and edit a BASIC program. The source line pre-compiler, also part of the Program Constructor, translates your BASIC program source text lines into tokenized lines. The Program Executor is used to execute the tokenized program — when you type RUN, the Program Executor takes over.

Both the Program Constructor and the Program Executor are designed to use data tables. Some of these tables are already contained in BASIC’s ROM (read-only memory). Others are constructed by BASIC in the user RAM (random-access memory). Understanding these various tables is an important key to understanding the design of Atari BASIC.

Tokens

In Atari BASIC, tokens are the intermediate code into which the source text is translated. They represent source-language symbols that come in various lengths — some as long as 100 characters (a long variable name) and others as short as one character ("+", "-" or "_`). Every token, however, is exactly one eight-bit byte in length.

Since most BASIC Language Symbols are more than one character long, the representation of a multi-character BASIC Language Symbol with a single-byte token can mean a considerable saving of program storage space.

A single-byte token symbol is also easier for the Program Executor to recognize than a multi-character symbol, since it can be evaluated by machine language routines much more quickly. The SEARCH routine — 76 bytes long — located at $A462 is a good example of how much assembly language it takes to recognize a multi-character symbol. On the other hand, the two instructions located at $AB42 are enough to
determine if a one-byte token is a variable. Because routines to recognize Atari BASIC’s one-byte tokens take so much less machine language, they execute relatively quickly.

The 256 possible tokens are divided into logical numerical groups that also make them simpler to deal with in assembly language. For example, any token whose value is 128 ($80) or greater represents a variable name. The logical grouping of the token values also means faster execution speeds, since, in effect, the computer only has to check bit 7 to recognize a variable.

The numerical grouping of the tokens is shown below:

<table>
<thead>
<tr>
<th>Token Value (Hex)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00-0D</td>
<td>Unused</td>
</tr>
<tr>
<td>0E</td>
<td>Floating Point Numeric Constant. The next six bytes will hold its value.</td>
</tr>
<tr>
<td>0F</td>
<td>String Constant. The next byte is the string length. A string of that length follows.</td>
</tr>
<tr>
<td>10-3C</td>
<td>Operators. See table starting at $A7E3 for specific operators and values.</td>
</tr>
<tr>
<td>3D-54</td>
<td>Functions. See table starting at $A820 for specific functions and values.</td>
</tr>
<tr>
<td>55-7F</td>
<td>Unused.</td>
</tr>
<tr>
<td>80-FF</td>
<td>Variables.</td>
</tr>
</tbody>
</table>

In addition to the tokens listed above, there is another set of single-byte tokens, the Statement Name Tokens. Every statement in BASIC starts with a unique statement name, such as LET, PRINT, and POKE. (An assignment statement such as "A = B + C," without the word LET, is considered to begin with an implied LET.) Each of these unique statement names is represented by a unique Statement Name Token.

The Program Executor does not confuse Statement Name Tokens with the other tokens because the Statement Name Tokens are always located in the same place in every statement — at the beginning. The Statement Name Token value is derived from its entry number, starting with zero, in the Statement Name Table at $A4AF.
Chapter Two

Tables
A table is a systematic arrangement of data or information. Tables in Atari BASIC fall into two distinct types: tables that are part of the Atari BASIC ROM and tables that Atari BASIC builds in the user RAM area.

ROM Tables
The following is a brief description of the various tables in the Atari BASIC ROM. The detailed use of these tables will be explained in subsequent chapters.

**Statement Name Table ($A4AF).** The first two bytes in each entry point to the information in the Statement Syntax Table for this statement. The rest of the entry is the name of the statement name in ATASCII. Since name lengths vary, the last character of the statement name has the most significant bit turned on to indicate the end of the entry. The value of the Statement Name Token is derived from the relative (from zero) entry number of the statement name in this table.

**Statement Execution Table ($AA00).** Each entry in this table is the two-byte address of the 6502 machine language code which will simulate the execution of the statement. This table is organized with the statements in the same order as the statements in the Statement Name Table. Therefore, the Statement Name Token can be used as an index to this table.

**Operator Name Table ($A7E3).** Each entry comprises the ATASCII text of an Operator Symbol. The last character of each entry has the most significant bit turned on to indicate the end of the entry. The relative (from zero) entry number, plus 16 ($10), is the value of the token for that entry. Each of the entries is also given a label whose value is the value of the token for that symbol. For example, the ";" symbol at $A7E8 is the fifth (from zero) entry in the table. The label for the ";" token is esc, and the value of ese is $15, or 21 decimal (16 + 5).

**Operator Execution Table ($AA70).** Each two-byte entry points to the address, minus one, of the routine which simulates the execution of an operator. The token value, minus 16, is used to access the entries in this table during execution time. The entries in this table are in the same order as in the Operator Name Table.

**Operator Precedence Table ($AC3F).** Each entry represents the relative execution precedence of an individual operator. The table entries are accessed by the operator tokens,
minus 16. Entries correspond with the entries in the Operator Name Table. (See Chapter 7.)

**Statement Syntax Table ($A60D).** Entries in this table are used in the process of translating the source program to tokens. The address pointer in the first part of each entry in the Statement Name Table is used to access the specific syntax information for that statement in this table. (See Chapter 5.)

**RAM Tables**
The tables that BASIC builds in the user RAM area will be explained in detail in Chapter 3. The following is a brief description of these tables:

- **Variable Name Table.** Each entry contains the source ATASCII text for the corresponding user variable symbol in the program. The relative (from zero) entry number of each entry in this table, plus 128, becomes the value of the token representing the variable.

- **Variable Value Table.** Each entry either contains or points to the current value of a variable. The entries are accessed by the token value, minus 128.

- **Statement Table.** Each entry is one tokenized BASIC program line. The tokenized lines are kept in this table in ascending numerical order by line number.

- **Array/String Table.** This table contains the current values for all strings and numerical arrays. The location of the specific values for each string and/or array variable is accessed from information in the Variable Value Table.

- **Runtime Stack.** This is the LIFO Runtime Stack, used to control the execution of GOSUB/RETURN and similar statements.

**Pre-compiler**
Atari BASIC translates the BASIC source lines from text to tokens as soon as they are entered. To do this, Atari BASIC must recognize the symbols of the BASIC Language. BASIC also requires that its symbols be combined in certain specific patterns. If the symbols don’t follow the required patterns, then Atari BASIC cannot translate the line. The process of checking a source line for the required symbol patterns is called **syntax checking.**

BASIC performs syntax checking as part of the tokenizing process. When the Program Editor receives a completed line of
input, the editor hands the line to the syntax routine, which
examines the first word of the line for a statement name. If a
valid statement name is not found, then the line is assumed to
be an implied LET statement.

The grammatical rules for each statement are contained in
the Statement Syntax Table. A special section of code examines
the symbols in the source line, under the direction of the
grammatical rules set forth in the Statement Syntax Table. If
the source line does not conform to the rules, then it is reported
back as an error. Otherwise, the line is translated to tokens.
The result of this process is returned to the Program Editor for
further processing.

Program Editor

When Atari BASIC is not executing statements, it is in the edit
mode. When the user enters a source line and hits return, the
editor accepts the line into a line buffer, where it is examined
by the pre-compiler. The pre-compiler returns either tokens or
an error text line.

If the line started with a line number, the editor inserts the
tokenized line into the Statement Table. If the Statement Table
already contains a line with the same line number, then the old
line is removed from the Statement Table. The new line is then
inserted just after the statement with the next lower line
number and just before the statement with the next higher line
number.

If the line has no line number, the editor inserts the line at
the end of the Statement Table. It then passes control to the
Program Executor, which will carry out the statement(s) in the
line at the end of the Statement Table.

Program Executor

The Program Executor has a pointer to the statement that it is to
execute. When control is passed to the executor, the pointer
points to the direct (command) line at the end of the statement
table. If that statement causes some other line to be executed
(RUN, GOTO, GOSUB, etc.), the pointer is changed to the
new line. Lines continue to be executed as long as nothing
stops that execution (END, STOP, error, etc.). When the
program execution is stopped, the Program Executor returns
control to the editor.
Chapter Two

When a statement is to be executed, the Statement Name Token (the first code in the statement) directs the interpreter to the specific code that executes that statement. For instance, if that token represents the PRINT statement, the PRINT execution code is called. The execution code for each statement then examines the other tokens and simulates their operations.

**Execute Expression**

Arithmetic and logical expressions (\(A + B, \ C/D + E, \ F < G\), etc.) are simulated with the Execute Expression code. Expression operators (\(+, - , *\), etc.) have execution precedence — some operators must be executed before some others. The expression \(1 + 3 \times 4\) has a value of 13 rather than 16 because \(*\) had a higher precedence than \(+\). To properly simulate expressions, BASIC rearranges the expression with higher precedence first.

BASIC uses two temporary storage areas to hold parts of the rearranged expression. One temporary storage area, the Argument Stack, holds arguments — values consisting of constants, variables, and temporary values resulting from previous operator simulations. The other temporary storage area, the Operator Stack, holds operators. Both temporary storage areas are managed as Last-In/First-Out (LIFO) stacks.

**LIFO Stacks**

A LIFO (Last In/First Out) stack operates on the principle that the last object placed in the stack storage area will be the first object removed from it. If the letters A, B, C, and D, in that order, were placed in a LIFO stack, then D would be the first letter removed, followed by C, B, and A. The operations required to rearrange the expression using these stacks will be explained in Chapter 7.

BASIC also uses another LIFO stack, the Runtime Stack, in the simulation of statements such as GOSUB and FOR. GOSUB requires that BASIC remember where in the statement table the GOSUB was located so it will return to the right spot when RETURN is executed. If more than one GOSUB is executed before a RETURN, BASIC returns to the statement after the most recent GOSUB.
Chapter Three

Memory Usage

Many of BASIC's functions are controlled by a set of tables built in RAM not already occupied by BASIC or the Operating System (OS). Figure 3.1 is a diagram of memory use by both programs. Every time a BASIC programmer enters a statement, memory requirements for the RAM tables change. Memory use by the OS also varies. Different graphics modes, for example, require different amounts of memory.

These changing memory requirements are monitored, and this series of pointers keeps BASIC and the OS from overlaying each other in memory:

- High memory address (HMADR) at location $02E5
- Application high memory (APHM) at location $000E
- Low memory address (LMADR) at location $02E7

When a graphics mode requires larger screen space, the OS checks the application high memory address (APHM) that has been set by BASIC. If there is enough room for the new screen, the OS uses the upper portion of space and sets the pointer HMADR to the bottom of the screen to tell the application how much space the OS is now using.

BASIC builds its table toward high memory from low memory. The pointer to the lowest memory available to an application, called LMADR in the BASIC listing, is set by the OS to tell BASIC the lowest memory address that BASIC can use. When BASIC needs more room for one of its tables, BASIC checks HMADR. If there is enough room, BASIC uses the space and puts the highest address it has used into APHM for OS.

BASIC's operation consists primarily of building, reading, and modifying tables. Pointers to the RAM tables are kept in consecutive locations in zero page starting at $80. These tables are, in order,

- Multipurpose Buffer
- Variable Name Table
- Variable Value Table
- String/Array Table
BASIC reserves space for a buffer at LMADR. It then builds the tables contiguously (without gaps), starting at the top of the buffer and extending as far as necessary towards APHM. When a new entry needs to be added to a table, all data in the tables above is moved upward the exact amount needed to fit the new entry into the right place.

**Figure 3-1. Memory Usage**
Chapter Three

Variable Name Table
The Variable Name Table (VNT) is built during the pre-compile process. It is read, but not modified, during execution — but only by the LIST statement. The VNT contains the names of the variables used in the program in the order in which they were entered.

The length of entries in the Variable Name Table depends on the length of the variable name. The high order bit of the last character of the name is on. For example, the ATASCII code for the variable name ABC is 41 42 43 (expressed in hexadecimal). In the Variable Name Table it looks like this:

41 42 C3

The $ character of a string name and the ( character of an array element name are stored as part of the variable name. The table entries for variables C, AA$, and X(3) would look like this:

C C3
AA$ 41 41 A4
X(3) 58 A8

It takes only two bytes to store X(3) because this table stores only X().

A variable is represented in BASIC by a token. The value of this token is the position (relative to zero) of the variable name in the Variable Name Table, plus $80. BASIC references an entry in the table by using the token, minus $80, as an index.

The Variable Name Table is not changed during execution time. The zero page pointer to the Variable Name Table is called VNTP in the BASIC listing.

Variable Value Table
The Variable Value Table (VVT) is also built during the pre-compile process. It is both read and modified during execution. There is a one-to-one correspondence in the order of entries between the Variable Name Table and the Variable Value Table. If XXX is the fifth variable in the Variable Name Table, then XXX's value is the fifth entry in the Variable Value Table. BASIC references a table entry by using the variable token, minus $80, as an index.

Each entry in the Variable Value Table consists of eight bytes. The first two bytes have the following meaning:
Chapter Three

1 2

<table>
<thead>
<tr>
<th>type</th>
<th>vnum</th>
</tr>
</thead>
</table>

\textit{type} = one byte, which indicates the type of variable

$\$00$ $\text{for floating point variable}$

$\$40$ $\text{for array variable}$

$\$80$ $\text{for string variable}$

\textit{vnum} = one byte, which indicates the relative position of the variable in the tables

The meaning of the next six bytes varies, depending on the type of variable (floating point, string, or array). In all three cases, these bytes are initialized to zero during syntaxing and during the execution of the RUN or CLR.

When the variable is a floating point number, the six bytes represent its value.

When the variable is an array, the remaining six bytes have the following format:

\begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 & 6 \\
\hline
\text{disp} & \text{dim1} & \text{dim2} \\
\end{array}

\textit{disp} = the two-byte displacement into string/array space of this array variable

\textit{dim1} = two bytes indicating the first dimension value

\textit{dim2} = two bytes indicating the second dimension value

All three of these values are set appropriately when the array is DIMensioned during execution.

When the variable is a string, the remaining six bytes have the following meaning:

\begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 & 6 \\
\hline
\text{disp} & \text{curl} & \text{maxl} \\
\end{array}
disp = the two-byte displacement into string/array space of
this string variable. This value is set when the string is
DIMensioned during execution.

curl = the two-byte current length of the string. This value
changes as the length of the string changes during
execution.

maxl = the two-byte maximum possible length of this string.
This value is set to the DIM value during execution.

When either a string or an array is DIMensioned during
execution, the low-order bit in the type byte is turned on, so
that the array type is set to $41 and the string type to $81.

The zero page pointer to the Variable Value Table is called
VVTP in the BASIC listing.

Statement Table

The Statement Table, built as each statement is entered during
editing, contains tokenized forms of the statements that were
entered. This table determines what happens during
execution.

The format of a Statement Table entry is shown in Figure
3-2. There can be several tokens per statement and several
statements per line.

Figure 3-2. Format of a Statement Table Entry

\[ \begin{array}{cccccccc}
\text{lnum} & \text{llen} & \text{slen} & \text{snt} & \text{toks} & \text{eos} & \text{slen} & \text{snt} & \text{toks} & \text{eos} & \text{eol} \\
\end{array} \]

\text{lnum} = the two-byte line number (low-order, high-order)
\text{llen} = the one-byte line length (the displacement to the next
line in the table)
\text{slen} = the one-byte statement length (the displacement to
the next statement in the line)
\text{snt} = the one-byte Statement Name Token
\text{toks} = the other tokens that make up the statement (this
is variable in length)
\text{eos} = the one-byte end of statement token
\text{eol} = the one-byte end of line token

The zero page pointer to the Statement Table is called
STMTAB in the BASIC listing.
Chapter Three

String/Array Table

The String/Array Table (also called String/Array Space) is created and modified during execution. Strings and arrays can be intermixed in the table, but they have different formats. Each array or string is pointed to by an entry in the Variable Value Table. The entry in the String/Array Table is created when the string or array is DIMensioned during execution. The data in the entry changes during execution as the value of the string or an element of the array changes.

An entry in the String/Array Table is not initialized to any particular value when it is created. The elements of arrays and the characters in a string cannot be counted upon to have any particular value. They can be zero, but they can also be garbage — data previously stored at those locations.

Array Entry

For an array, the String/Array Table contains one six-byte entry for each array element. Each element is a floating point number, stored in raveled order. For example, the entry in the String/Array Table for an array that was dimensioned as A(1,2) contains six elements, in this order:

\[ A(0,0) \quad A(0,1) \quad A(0,2) \quad A(1,0) \quad A(1,1) \quad A(1,2) \]

String Entry

A string entry in the String/Array Table is created during execution, when the string is DIMensioned. The size of the entry is determined by the DIM value. The "value" of the string to BASIC at any time is determined by the data in the String/Array Table and the current length of the string as set in the Variable Value Table.

The zero page pointer to the String/Array Table is called STARP in the BASIC listing.

The Runtime Stack is created during execution. BASIC uses this LIFO stack to control processing of FOR/NEXT loops and GOSUBs. When either a FOR or a GOSUB statement is encountered during execution, an entry is put on the Runtime Stack. When a NEXT, RETURN, or a POP statement is encountered, entries are pulled off the stack.

Both the FOR entry and the GOSUB entry have a four-byte header:
type = one byte indicating the type of element
  GOSUB type = 0
  FOR type = non-zero
lnum = the two-byte number of the line which contains the statement (low-order, high-order)
disp = one byte indicating the displacement into the line in the Statement Table of the token which caused this stack entry.

The FOR-type byte is actually the token representing the loop control variable from the FOR statement. (In the statement FOR I = 1 to 10, I is the loop control variable.) So the FOR-type byte will have a value of $80$ through $$FF$ — the possible values of a variable token.

The FOR entry contains 12 additional bytes, formatted like this:

```
1 2 3 4 5 6 7 8 9 10 11 12
```

\begin{align*}
  sval &= \text{the six-byte (floating point) limit value at which to stop the loop} \\
  step &= \text{the six-byte (floating point) STEP value to increment by}
\end{align*}

The GOSUB entry consists entirely of the four-byte header. The LIST and READ statements also put a GOSUB type entry on the Runtime Stack, so that the line containing the LIST or READ can be found again when the statement has finished executing.

The zero page pointer to the Runtime Stack is called RUNSTK in the BASIC listing.
Chapter Three

Zero Page Table Pointers
The starting addresses of the tables change dynamically during both program construction and program execution. BASIC keeps the current start addresses of the tables and other pointers required to manage memory space in contiguous zero-page cells. Each pointer is a two-byte address, low byte first.

Since these zero page cell addresses remain constant, BASIC is always able to find the tables. Here are the zero page pointers used in memory management, their names in the BASIC listing, and their addresses:

<table>
<thead>
<tr>
<th>Name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multipurpose Buffer</td>
<td>$80, $81</td>
</tr>
<tr>
<td>Variable Name Table</td>
<td>VNT</td>
</tr>
<tr>
<td>VNT dummy end</td>
<td>VNTD</td>
</tr>
<tr>
<td>Variable Value Table</td>
<td>VVT</td>
</tr>
<tr>
<td>Statement Table</td>
<td>STMTAB</td>
</tr>
<tr>
<td>Current Statement Pointer</td>
<td>STMCUR</td>
</tr>
<tr>
<td>String/Array Table</td>
<td>STARP</td>
</tr>
<tr>
<td>Runtime Stack</td>
<td>RUNSTK</td>
</tr>
<tr>
<td>Top of used memory</td>
<td>MEMTOP</td>
</tr>
</tbody>
</table>

Memory Management Routines
Memory Management routines allocate space to the BASIC tables as needed. There are two routines: expand, to add space, and contract, to delete space. Each routine has one entry point for cases in which the number of bytes to be added or deleted is less than 256, and another when it is greater than or equal to 256.

The EXPAND and CONTRACT routines often move many thousands of bytes each time they are called. The 6502 microprocessor is designed to move fewer than 256 bytes of data very quickly. When larger blocks of data are moved, the additional 6502 instructions required can make the process very slow. The EXPAND and CONTRACT routines circumvent this by using the less-than-256-byte fast-move capabilities in the movement of thousands of bytes. The end result is a set of very fast and very complex data movement routines.

All of this complexity does have a drawback. The infamous Atari BASIC lock-up problem lives in these two routines. If an EXPAND or CONTRACT requires that an exact multiple of 256 bytes be moved, then the routines move things from the wrong
place in memory to the wrong place in memory, whereupon the computer locks up and won’t respond. The only way to avoid losing hours of work this way is to SAVE to disk or cassette frequently.

**EXPAND ($A881)**

Parameters at entry:

```
gerest
    X = the zero page address containing the pointer to the location after which space is to be added
    Y = the low-order part of the number of bytes to expand
    A = the high-order part of the number of bytes to expand
```

The routine creates a hole in the table memory, starting at a requested location and continuing the requested number of bytes.

The routine first checks to see that there is enough free memory space to satisfy the request. It adds the requested expand size to each of the zero-page table pointers between the one pointed to by the X register and MEMTOP. Then each pointer will point to the correct address when EXPAND is done.

EXPAND then creates space at the address indicated by the X register. The number of bytes required is contained in the Y and A registers. (Y contains the least significant byte, while A contains the most significant.) All data from the requested address to the address pointed to by MEMTOP is moved toward high memory by the requested number of bytes. This creates a hole of the proper size.

The routine then sets Application High Memory (APHM) to the value in MEMTOP. This tells the OS the highest memory address that BASIC is currently using.

**EXPLOW ($A87F)**

Parameters at entry:

```
gerest
    X = zero page address containing the pointer to the location after which space is to be added
    Y = number of bytes to expand (low-order byte only)
```
Chapter Three

This is an additional entry point for the EXPAND routine. It is used when the number of bytes to be added to the table is less than 256.

This routine first loads the 6502 accumulator with zero to indicate the most significant byte of the expand length. It then functions exactly like EXPAND.

**CONTRACT ($A8FD)**
Parameters at entry:

\[
\begin{align*}
\text{register} \\
X & = \text{zero page address containing the pointer to the starting location where space is to be removed} \\
Y & = \text{the low-order part of the number of bytes to contract} \\
A & = \text{the high-order part of the number of bytes to contract}
\end{align*}
\]

This routine removes a requested number of bytes at a requested location by moving all the data from higher in the tables downward the exact amount needed to replace the unwanted bytes.

It subtracts the requested contract size from each of the zero page table pointers between the one pointed to by the X register and MEMTOP. Then each pointer will point to the correct address when CONTRACT is done.

The routine sets application high memory (APHM) to the value in MEMTOP to indicate to the OS the highest memory address that BASIC is currently using.

The block of data to be moved downward is defined by starting at the address pointed to by the zero-page address pointed to in X, plus the offset number stored in Y and A, and then continuing to the address specified at MEMTOP. Each byte of data in that block is moved downward in memory by the number of bytes specified in Y and A, effectively erasing all the data between the specified address and that address plus the requested offset.

**CONTLOW ($A8FB)**
Parameters at entry:

\[
\begin{align*}
\text{register} \\
X & = \text{the zero page address containing the pointer to the location at which space is to be removed}
\end{align*}
\]

22
Y = the number of bytes to contract (low-order byte only)

This routine is used to remove fewer than 256 bytes from the tables at a requested location by moving all the data from higher in the tables downward the exact amount needed to replace the unwanted bytes.

This routine first loads the 6502 accumulator with zero to serve as the most significant byte of the contract length. It then functions exactly like CONTRACT.

**Miscellaneous Memory Allocations**

Besides the tables, which change dynamically, BASIC also uses buffers and stacks at fixed locations.

The Argument/Operator Stack is allocated at BASIC's low memory address and occupies 256 bytes. During pre-compiling it is used as the output buffer for the tokens. During execution, it is used while evaluating an expression. This buffer/stack is referenced by a pointer at location $80. This pointer has several names in the BASIC listing: LOMEM, ARGOPS, ARGSTK, and OUTBUFF.

The Syntax Stack is used during the process of syntaxing a statement. It is referenced directly — that is, not through a pointer. It is located at $480 and is 256 bytes long.

The Line Buffer is the storage area where the statement is placed when it is ENTERed. It is the input buffer for the edit and pre-compile processes. It is 128 bytes long and is referenced directly as LBUFF. Often the address of LBUFF is also put into INBUFF so that the buffer can be referenced through a pointer, though INBUFF can point to other locations during various phases of BASIC's execution.
Chapter Four

Program Editor

The Atari keyboard is the master control panel for Atari BASIC. Everything BASIC does has its origins at this control panel. The Program Editor’s job is to service the control panel and respond to the commands that come from it.

The editor gets a line from the user at the keyboard; does some preliminary processing on the line; passes the line to the pre-compiler for further processing; inserts, deletes, or replaces the line in the Statement Table; calls the Program Executor when necessary; and then waits to receive the user’s next line input.

Line Processing

The Program Editor, which starts at $A060, begins its process by resetting the 6502 CPU stack. Resetting the CPU stack is a drastic operation that can only occur at the beginning of a logical process. Each time Atari BASIC prepares to get a new line from the user, it restarts its entire logical process.

Getting a Line

The Program Editor gets a user’s line by calling CIO. The origin of the line is transparent to the Program Editor. The line may have been typed in at the keyboard or entered from some external device like the disk (if the ENTER command was given). The Program Editor simply calls CIO and asks it to put a line of not more than 255 bytes into the buffer pointed to by INBUFF ($F3). INBUFF points to the 128-byte area defined at LBUFF ($580).

The OS’s screen editor, which is involved in getting a line from the keyboard, will not pass BASIC a line that is longer than 120 bytes. Normally, then, the 128-byte buffer at LBUFF is big enough to contain the user’s line.

Sometimes, however, if a line was originally entered from the keyboard with few blanks and many abbreviations, then LISTed to and re-ENTERed from the disk, an input line may be longer than 128 bytes. When this happens, data in the $600 page is overlaid. A LINE TOO LONG error will not necessarily
occur at this point. A LINE TOO LONG error occurs only if the Pre-compiler exceeds its stack while processing the line or if the tokenized line OUTBUFF exceeds 256 bytes. These overflows depend on the complexity of the line rather than on its actual length.

When CIO has put a line into the line buffer (LBUFF) and the Program Editor has regained control, it checks to see if the user has changed his mind and hit the break key. If the user did indeed hit break, the Program Editor starts over and asks CIO for another line.

**Flags and Indices**

In order to help control its processing, the Program Editor uses flags and indices. These must be given initial values.

**CIX and COX.** The index CIX ($F2) is used to access the user’s input line in the line buffer (LBUFF), while COX ($94) is used to access the tokenized statement in the output buffer (OUTBUFF). These buffers and their indices are also used by the pre-compiler. The indices are initialized to zero to indicate the beginning of the buffers.

**DIRFLG.** This flag byte ($A6) is used by the editor to remember whether a line did or did not have a line number, and also to remember if the pre-compiler found an error in that line. DIRFLG is initialized to zero to indicate that the line has a line number and that the pre-compiler has not found an error.

**MAXCIX.** This byte ($9F) is maintained in case the line contains a syntax error. It indicates the displacement into LBUFF of the error. The character at this location will then be displayed in inverse video. The Program Editor gives this byte the same initial value as CIX, which is zero.

**SVVNTP.** The pointer to the current top of the Variable Name Table (VNTD) is saved as SVVNTP ($AD) so that if there is a syntax error in this line, any variables that were added can be removed. If a user entered an erroneous line, such as 100 A=XAND B, the variable XAND would already have been added to the variable tables before the syntax error was discovered. The user probably meant to enter 100 A=X AND B, and, since there can only be 128 variables in BASIC, he probably does not want the variable XAND using up a place in the variable tables. The Program Editor uses SVVNTP to find the entry in the Variable Name Table so it can be removed.
SVVVTE. The process used to indicate which variable entries to remove from the Variable Value Table in case of error is different. The number of new variables in the line (SVVVTE,$B1) is initialized to zero. The Program Pre-compiler increments the value every time it adds a variable to the Variable Value Table. If a syntax error is detected, this number is multiplied by eight (the number of bytes in each entry on the Variable Value Table) to get the number of bytes to remove, counting backward from the most recent value entered.

Handling Blanks
In many places in the BASIC language, blanks are not significant. For example,

```
100 IF X = 6 THEN GOTO 500
```

has the same meaning as

```
100 IF X = 6 THEN GOTO 500.
```

The Program Editor, using the SKIPBLANK routine ($DBA1), skips over unnecessary blanks.

Processing the Line Number
Once the editor has skipped over any leading blanks, it begins to examine the input line, starting with the line number. The floating point package is called to determine if a line number is present, and, if so, to convert the ATASCII line number to a floating point number. The floating point number is converted to an integer, saved in TSLNUM for later use, and stored in the tokenized line in the output buffer (OUTBUFF).

The routine used to store data into OUTBUFF is called :SETCODE ($A2C8). When :SETCODE stores a byte into OUTBUFF, it also increments COX, that buffer’s index.

BASIC could convert the ATASCII line number directly to an integer, but the routine to do this would not be used any other time. Routines to convert ATASCII to floating point and floating point to integer already exist in BASIC for other purposes. Using these existing routines conserves ROM space.

An interesting result of this sequence is that it is valid to enter a floating point number as a line number. For example, 100.1, 10.9, or 2.05E2 are valid line numbers. They would be converted to 100, 11, and 205 respectively.

If the input line does not start with a line number, the line is considered to be a direct statement. DIRFLG is set to $80 so
that the editor can remember this fact. The line number is set to 32768 ($8000). This is one larger than the largest line number a user is allowed to enter. BASIC later makes use of this fact in processing the direct statement.

**Line length.** The byte after the line number in the tokenized line in OUTBUFF is reserved so that the line length (actually the displacement to the next line) can be inserted later. (See Chapter 2.) The routine :SETCODE is called to reserve the byte by incrementing (COX) to indicate the next byte.

**Saving erroneous lines.** In the byte labeled STMSTRT, the Program Editor saves the index into the line buffer (LBUFF) of the first non-blank character after the line number. This index is used only if there is a syntax error, so that all the characters in the erroneous line can be moved into the tokenized line buffer and from there into the Statement Table.

There are advantages to saving an erroneous line in the Statement Table, because you can LIST the error line later. The advantage is greatest, not when entering a program at the keyboard, but when entering a program originally written in a different BASIC on another machine (via a modem, perhaps). Then, when a line that is not correct in Atari BASIC is entered, the line is flagged and stored — not discarded. The user can later list the program, find the error lines, and re-enter them with the correct syntax for Atari BASIC.

**Deleting lines.** If the input line consists solely of a line number, the Program Editor deletes the line in the Statement Table which has that line number. The deletion is done by pointing to the line in the Statement Table, getting its length, and calling CONTRACT. (See Chapter 3.)

**Statement Processing**

The user's input line may consist of one or more statements. The Program Editor repeats a specific set of functions for each statement in the line.

**Initializing**

The current index (COX) into the output buffer (OUTBUFF) is saved in a byte called STMLBD. A byte is reserved in OUTBUFF by the routine :SETCODE. Later, the value in
STMLBD will be used to access this byte, and the statement length (the displacement to the next statement) will be stored here.

**Recognizing the Statement Name**

After the editor calls SKBLANK to skip blanks, it processes the statement name, now pointed to by the input index (CIX). The editor calls the routine SEARCH ($A462) to look for this statement name in the Statement Name Table. SEARCH saves the table entry number of this statement name into location STENUM.

The entry number is also the Statement Name Token value, and it is stored into the tokenized output buffer (OUTBUFF) as such by :SETCODE. The SEARCH routine also saves the address of the entry in SRCADR for use by the pre-compiler.

If the first word in the statement was not found in the Statement Name Table, the editor assumes that the statement is an implied LET, and the appropriate token is stored. It is left to the pre-compiler to determine if the statement has the correct syntax for LET.

The editor now gives control to the pre-compiler, which places the appropriate tokens in OUTBUFF, increments the indices CIX and COX to show current locations, and indicates whether a syntax error was detected by setting the 6502 carry flag on if there was an error and clearing the carry flag if there was not. (See Chapter 5.)

**If a Syntax Error Is Detected**

If the 6502 carry flag is set when the editor regains control, the editor does error processing.

In MAXCIX, the pre-compiler stored the displacement into LBUFF at which it detected the error. The Program Editor changes the character at this location to inverse video.

The character in inverse video may not be the point of error from your point of view, but it is where the pre-compiler detected an error. For example, assume you entered X = YAND Z. You probably meant to enter X = Y AND Z, and therefore would consider the error to be between Y and AND. However, since YAND is a valid variable name, X = YAND is a valid BASIC statement.

The pre-compiler doesn't know there is an error until it encounters B. The value of highlighting the error with inverse video
video is that it gives the user an approximation of where the error is. This can be a big advantage, especially if the input line contained multiple statements or complex expressions.

The next thing the editor does when a syntax error has been detected is set a value in DIRFLG to indicate this fact for future reference. Since the DIRFLG byte also indicates whether this is a direct statement, the error indicator of $40$ is ORed with the value already in DIRFLG.

The editor takes the value that it saved in STMSTRT and puts it into CIX so that CIX now points to the start of the first statement in the input line in LBUFF. STMLBD is set to indicate the location of the first statement length byte in OUTBUFF. (A length will be stored into OUTBUFF at this displacement at a later time.)

The editor sets the index into OUTBUFF (COX) to indicate the Statement Name Token of the first statement in OUTBUFF, and stores a token at that location to indicate that this line has a syntax error. The entire line (after the line number) is moved into OUTBUFF. At this point COX indicates the end of the line in OUTBUFF. (Later, the contents of OUTBUFF will be moved to the Statement Table.)

This is the end of the special processing for an erroneous line. The process that follows is done for both correct and erroneous lines.

**Final Statement Processing**
During initial line processing, the Program Editor saved in STMLBD a value that represents the location in OUTBUFF at which the statement length (displacement to the next statement) should be stored. The Program Editor now retrieves that value from STMLBD. Using this value as an index, the editor stores the value from COX in OUTBUFF as the displacement to the next statement.

The Program Editor checks the next character in LBUFF. If this character is not a carriage return (indicating end of the line), then the statement processing is repeated. When the carriage return is found, COX will be the displacement to the next line. The Program Editor stores COX as the line length at a displacement of two into OUTBUFF.
Statement Table Processing
The final tokenized form of the line exists in OUTBUFF at this point. The Program Editor’s next task is to insert or replace the line in the Statement Table.

The Program Editor first needs to create the correct size hole in the Statement Table. The editor calls the GETSTMT routine ($A9A2) to find the address where this line should go in the Statement Table. If a line with the same line number already exists, the routine returns with the address in STMCUR and with the 6502 carry flag off. Otherwise, the routine puts the address where the new line should be inserted in the Statement Table into STMCUR and turns on the 6502 carry flag. (See Chapter 6.)

If the line does not exist in the Statement Table, the editor loads zero into the 6502 accumulator. If the line does exist, the editor calls the GETLL routine ($A9DD) to put the line length into the accumulator. The editor then compares the length of the line already in the Statement Table (old line) with the length of the line in OUTBUFF (new line).

If more room is needed in the Statement Table, the editor calls the EXPLOW ($A87F; see Chapter 3). If less space is needed for the new line, it calls a routine to point to the next line (GNXTL, at location $A9D0; see Chapter 6), and then calls the CONTLOW ($A8FB; see Chapter 3).

Now that we have the right size hole, the tokenized line is moved from OUTBUFF into the Statement Table at the location indicated by STMCUR.

Line Wrap-up
After the line has been added to the Statement Table, the editor checks DIRFLG for the syntax error indicator. If the second most significant bit ($40) is on, then there is an error.

Error Wrap-up
If there is an error, the editor removes any variables that were added by this line by getting the number of bytes that were added to the Variable Name Table and the Variable Value Table from SVVNTP and SVVVTE. It then calls CONTRACT ($A8FD) to remove the bytes from each table.

Next, the editor lists the line. The Statement Name Token, which was set to indicate an error, causes the word "ERROR"
to be printed. An inverse video character indicates where the error was detected. The editor now waits for you to enter another line.

Handling Correct Lines
If the line was syntactically correct, the editor again examines DIRFLG. In earlier processing, the most significant bit ($80) of this byte was set on if the line was a direct statement. If it is not a direct statement, then the editor is finished with the line, and it waits for another input line.

If the line is a direct statement, earlier processing already assigned it a line number of 32768 ($8000), one larger than the largest line number a user can enter. Since lines are arranged in the Statement Table in ascending numerical order, this line will have been inserted at the end of the table. The current statement pointer (STMCUR—$8A, $8B) points to this line.

The Program Editor transfers control to a Program Executor routine, Execution Control (EXECNL at location $A95F), which will handle the execution of the direct statement. (See Chapter 6.)
The Pre-compiler

The symbols and symbol-combining rules of Atari BASIC are coded into Syntax Tables, which direct the Program Pre-compiler in examining source code and producing tokens. The information in the Syntax Tables is a transcription of a meta-language definition of Atari BASIC.

The Atari BASIC Meta-language
A meta-language is a language which describes or defines another language. Since a meta-language is itself a language, it also has symbols and symbol-combining rules — which define with precision the symbols and symbol-combining rules of the subject language.

Atari BASIC is precisely defined with a specially developed meta-language called the Atari BASIC Meta-language, or ABML. (ABML was derived from a commonly used compiler-technology meta-language called BNF.) The symbols and symbol-combining rules of ABML were intentionally kept very simple.

Making Up a Language
To show you how ABML works, we’ll create an extremely simple language called SAP, for Simple Arithmetic Process. SAP symbols consist of variables, constants, and operators.

- Variables: The letters A, B, and C only.
- Constants: The numbers 1,2,3,4,5,6,7,8, and 9 only.
- Operators: The characters +, -, *, /, and ! only. Of course, you already know the functions of all the operators except ‘!’ ‘.’. The character ‘!’ is a pseudo-operator of the SAP language used to denote the end of the expression, like the period that ends this sentence.

The grammar of the SAP language is precisely defined by the ABML definition in Figure 5-1.
The ABML symbols used to define the SAP language in Figure 5-1 are:

\[
\begin{align*}
\text{SAP} & := <\text{expression}>!
\text{<expression>} & := <\text{value}> <\text{operation}>|
\text{<operation>} & := <\text{constant}> <\text{variable}>
\text{<constant>} & := 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
\text{<variable>} & := A | B | C
\text{<operator>} & := + | - | * | /
\end{align*}
\]

The ABML symbols used to define the SAP language in Figure 5-1 are:

\[
\begin{align*}
:= & \quad \text{is defined as} \quad \text{Whatever is on the left of : = is defined as consisting of whatever is on the right of : =, and in that order.}
\mid & \quad \text{or} \quad \text{The symbol | allows choices for what something is defined as. For instance, in the sixth line <variable> can be A or B or C. If | does not appear between two symbols, then there is no choice. For example, in the second line <expression> must have both <value> and <operation>, in that order, to be valid.}
< > & \quad \text{label} \quad \text{Whatever comes between < and > is an ABML label. All labels, as non-terminal symbols, must be defined at some point, though the definitions can be circular notice that <operation> is part of the definition of <expression> in the second line, while in the third line <expression> is part of the definition of <operation>.
\text{terminal} & \quad \text{symbols} \quad \text{Symbols used in definitions, which are not enclosed by < and > and are also not one of the ABML symbols, are terminal symbols in the language being defined by ABML. In SAP, some terminal symbols are A, !, B, *, and 1. They cannot be defined as consisting of other symbols they are themselves the symbols that the SAP language manipu-}
\end{align*}
\]
lates, and must appear exactly as they are shown to be valid in SAP. In effect, they are the vocabulary of the SAP language.

**Statement Generation**

The ABML description of SAP can be used to generate grammatically correct statements in the SAP language. To do this, we merely start with the first line of the definition and replace the non-terminal symbols with the definitions of those symbols. The replacement continues until only terminal symbols remain. These remaining terminal symbols constitute a grammatically correct SAP statement.

Since the *or* statement requires that one and only one of the choices be used, we will have to arbitrarily replace the non-terminal with the one valid choice.

Figure 5-2 illustrates the ABML statement generation process.

**Figure 5-2. The Generation of One Possible SAP Statement**

(1) \( \text{SAP} := <\text{expression}>! \)
(2) \( \text{SAP} := <\text{value}> <\text{operation}>! \)
(3) \( \text{SAP} := <\text{variable}> <\text{operation}>! \)
(4) \( \text{SAP} := B <\text{operation}>! \)
(5) \( \text{SAP} := B <\text{operator}> <\text{expression}>! \)
(6) \( \text{SAP} := B^* <\text{expression}>! \)
(7) \( \text{SAP} := B^* <\text{value}> <\text{operation}>! \)
(8) \( \text{SAP} := B^* <\text{constant}> <\text{operation}>! \)
(9) \( \text{SAP} := B^*4 <\text{operation}>! \)
(10) \( \text{SAP} := B^*4 <\text{operator}> <\text{expression}>! \)
(11) \( \text{SAP} := B^*4 + <\text{expression}>! \)
(12) \( \text{SAP} := B^*4 + <\text{value}> <\text{operation}>! \)
(13) \( \text{SAP} := B^*4 + <\text{variable}> <\text{operation}>! \)
(14) \( \text{SAP} := B^*4 + C <\text{operation}>! \)
(15) \( \text{SAP} := B^*4 + C! \)

In (2), \(<\text{value}> <\text{operation}>\) replaces \(<\text{expression}>\) because the ABML definition of SAP (Figure 5-1) defines \(<\text{expression}>\) as \(<\text{value}> <\text{operation}>.\)

In (3), the non-terminal \(<\text{value}>\) is replaced with
<variable> . The definition of <value> gives two choices for the substitution of <value> . We happened to choose <variable> .

In (4), we reach a terminal symbol, and the process of defining <value> ends. We happened to choose B to replace <variable> .

In (5), we go back and start defining <operation> . There are two choices for the replacement of <operation> , either <operator> <expression> or nothing at all (since there is nothing to the right of I in the second line of Figure 5-1). If nothing had been chosen, then (5) would have been: SAP := B!
The statement B! has no further non-terminals; the process would have been finished, and a valid statement would have been produced. Instead we happened to choose <operator> <expression> .

The SAP definition for <expression> is <value> <operation> . If we replace <operation> with its definition we get:

<expression> := <value> <operator> <expression>

The definition of <expression> includes <expression> as part of its definition. If the <operator> <expression> choice were always made for <operation> , then the process of replacement would never stop. A SAP statement can be infinitely long by definition. The only thing which prevents us from always having an infinitely long SAP statement is that there is a second choice for the replacement of <operation> : nothing.

The replacements in (5) and (10) reflect the repetitive choices of defining <expression> in terms of itself. The choice in (15) reflects the nothing choice and thus finishes the replacement process.

Computerized Statement Generation
If we slightly modify our procedure for generating statements, we will have a process that could be easily programmed into a computer. Instead of arbitrarily replacing the definition of non-terminals, we can think of the non-terminal as a GOSUB.

When we see <X> := <Y> <Z> , we can think of <Y> as being a subroutine-type procedure:

(a) Go to the line that has <Y> on the left side.
(b) Process the definition (right side) of <Y> .
(c) If while processing the definition of \(< Y >\), other non-terminals are found, GOSUB to them.
(d) If while processing the definition of \(< Y >\) we encounter a terminal, output the terminal symbol as the next symbol of the generated statement.
(e) When the definition of \(< Y >\) is finished, return to the place that \(< Y >\) was called from and continue.

Since ABML is structured so that it can be programmed, a fascinating exercise is to design a simple English sentence grammar with ABML, then write a BASIC program to generate valid English sentences at random. The randomness of the sentences would be derived by using the RND function to select from the definitions or choices. An example of such a grammar is shown in Figure 5-3. (The programming exercise is left to you.)

**Figure 5-3. A Simple English Sentence Grammar in ABML**

```
SENTENCE := <subject> <adverb> <verb> <object>.
<subject> := The <adjective> <noun>
<verb> := eats | sleeps | drinks | talks | hugs
<adverb> := quickly | silently | slowly | viciously | lovingly | sadly |
<object> := at home | in the car | at the table | at school | <subject>
<noun> := boy | girl | dog | programmer | computer | teacher
<adjective> := happy | sad | blue | light | round | smart | cool | nice |
```

**Syntactical Analysis**
The process of examining a language statement for grammatical correctness is called syntactical analysis, or syntaxing.

Statement verification is similar to statement generation. Instead of arbitrarily choosing which or definition to use, however, the choices are already made, and we must check to see whether the statement symbols are used in valid patterns. To do this, we must process through each or definition until we find a matching valid terminal symbol.

The result of statement generation is a valid, grammatically correct statement, but the result of statement verification is a
statement validity indication, which is a simple yes or no. Either the statement is grammatically correct or it is not. Failure occurs when some statement symbol cannot be matched with a valid terminal symbol under the rules of the grammar.

**The Reporting System**

To use the pass/fail result of statement verification, we must build a reporting system into the non-terminal checking process. Whenever we, in effect, GOSUB to a non-terminal definition, that non-terminal definition must report its pass/fail status.

A fail status is generated and returned by a non-terminal definition when it finds no matching terminal for the current statement symbol. If the current statement symbol is B and the <constant> definition in the SAP language is called, then <constant> would report a fail status to the routine that called it.

A pass status is returned when a terminal symbol is found which matches the current statement symbol. If our current statement symbol had been 7 instead of B, then <constant> would have reported pass.

Whenever such a match does occur, we return to the statement, and the next symbol to the right becomes the new current symbol for examination and verification.

**Cycling Through the Definitions**

In SAP, the <constant> definition is called from the <value> definition. If <constant> reports fail, then we examine the next or choice, which is <variable>. The current symbol is B, so <variable> reports pass.

Since at least one of the or choices of <value> has reported pass, <value> will report pass to its caller. If both <constant> and <variable> had reported fail, then <value> would report fail to its caller.

The caller of <value> is <expression>. If <value> reports pass, <operation> is called. If <operation> reports pass, then <expression> can report pass to its caller. If either <value> or <operation> reports fail, then <expression> must report fail, since there are no other or choices for <expression>.

The definition of <operation> contains a special pass/fail property. If either <operator> or <expression> reports fail,
then the or choice must be examined. In this case the or choice is nothing. The or nothing means something special: report pass, but do not advance to the next symbol.

The final pass/fail report is generated from the first line of the definition. If <expression> reports pass and the next symbol is !, then SAP reports pass. If either one of these conditions has a fail status, then SAP must report fail to whatever called SAP from outside the language.

**Backing Up**

Sometimes it is necessary to back up over symbols which have already been processed. Let's assume that there was a definition of the type <X> := <Y>|<Z>. It is possible that while <Y> is attempting to complete its definition, it will find a number of valid matching terminal symbols before it discovers a symbol that it cannot match. In this case, <Y> would have consumed a number of symbols before it decided to report fail. All of the symbols that <Y> consumed must be unconsumed before <Z> can be called, since <Z> will need to check those same symbols.

The process of unconsuming symbols is called backup. Backup is usually performed by the caller of <Y>, which remembers which source symbol was current when it called <Y>. If <Y> reports fail, then the caller of <Y> restores the current symbol pointer before calling <Z>.

**Locating Syntax Error**

When a final report of fail is given for a statement, it is often possible to guess where the error occurred. In a left-to-right system, the symbol causing the failure is usually the symbol which follows the rightmost symbol found to be valid. If we keep track of the rightmost valid symbol during the various backups, we can report a best guess as to where the failure-causing error is located. This is exactly what Atari BASIC does with the inverse video character in the ERROR line.

For simplicity, our example was coded for SAP, but the syntactical analysis we have just described is essentially the process that the Atari BASIC pre-compiler uses to verify the grammar of a source statement. The Syntax Tables are an ABML description of Atari BASIC. The pre-compiler, also known as the syntaxer, contains the routines which verify BASIC statements.
Chapter Five

Statement Syntax Tables

There is one entry in the Syntax Tables for each BASIC statement. Each statement entry in the Syntax Table is a transcription of an ABML definition of the grammar for that particular statement. The starting address of the table entry for a particular statement is pointed to by that statement's entry in the Statement Name Table.

The data in the Syntax Tables is very much like a computer machine language. The pseudo-computer which executes this pseudo-machine language is the pre-compiler code. Like any machine language, the pseudo-machine language of the Syntax Tables has instructions and instruction operands. For example, an ABML non-terminal symbol is transcribed to a code which the pre-compiler executes as a type of "GOSUB and report pass/fail" command.

Here are the pseudo-instruction codes in the Syntax Tables; each is one byte in length.

Absolute Non-Terminal Vector

Name: ANTV
Code: $00

This is one of the forms of the non-terminal GOSUB. It is followed by the address, minus 1, of the non-terminal's definition within the Syntax Table. The address is two bytes long, with the least significant byte first.

External Subroutine Call

Name: ESRT
Code: $01

This instruction is a special type of terminal symbol checker. It is followed by the address, minus 1, of a 6502 machine language routine. The address is two bytes long, with the least significant byte first. The ESRT instruction is a deus ex machina — the "god from the machine" who solved everybody's problems at the end of classical Greek plays. There are some terminals whose definition in ABML would be very complex and require a great many instructions to describe. In these cases, we go outside the pseudo-machine language of the Syntax Tables and get help from 6502 machine language routines — the deus ex machina that quickly gives the desired
result. A numeric constant is one example of where this outside help is required.

**ABML or**

Name: OR  
Value: $02

This is the familiar ABML or symbol ( | ). It provides for an alternative definition of a non-terminal.

**Return**

Name: RTN  
Value: $03

This code signals the end of an ABML definition line. When we write an ABML statement on paper, the end of a definition line is obvious — there is no further writing on the line. When ABML is transcribed to machine codes, the definitions are all pushed up against each other. Since the function that is performed at the end of a definition is a return, the end of definition is called return (RTN).

**Unused** (Codes $04 through $0D are unused.)

**Expression Non-Terminal Vector**

Name: VEXP  
Value: $0E

The ABML definition for an Atari BASIC expression is located at $A60D. Nearly every BASIC statement definition contains the possibility of having <expression> as part of it. VEXP is a single-byte call to <expression>, to avoid wasting the two extra bytes that ANTV would take. The pseudo-machine understands that this instruction is the same as an ANTV call to <expression> at $A60D.

**Change Last Token**

Name: CHNG  
Value: $0F

This instruction is followed by a one-byte change to token value. The operator token instructions cause a token to be placed into the output buffer. Sometimes it is necessary to change the token that was just produced. For example, there are several = operators. One = operator is for the assignment
Chapter Five

statement LET X = 4. Another = operator is for comparison operations like IF Y = 5. The pseudo-machine will generate the assignment = token when it matches = . The context of the grammar at that point may have required a comparison = token. The CHNG instruction rectifies this problem.

**Operator Token**

- **Name:** (many)
- **Value:** $10 through $7F

These instructions are terminal codes for the Atari BASIC Operators. The code values are the values of each operator token. The values, value names, and operator symbols are defined in the Operator Name Table (see Chapter 2).

When the pseudo-machine sees these terminal symbol representations, it compares the symbol it represents to the current symbol in the source statement. If the symbols do not match, then fail status is generated. If the symbols match, then pass status is generated, the token (instruction value) is placed in the token output buffer, and the next statement source symbol becomes the current symbol for verification.

**Relative Non-Terminal Vectors**

- **Name:** (none)
- **Value:** $80 — $BF (Plus)
  - $C0 — $FF (Minus)

This instruction is similar to ANTV, except that it is a single byte. The upper bit is enough to signal that this one-byte code is a non-terminal GOSUB. The destination address of the GOSUB is given as a position relative to the current table location. The values $80 through $BF correspond to an address which is at the current table address plus $00 through $3F. The values $C0 through $FF correspond to an address which is at the current table address minus $01 through $3F.

**Pre-compiler Main Code Description**

The pre-compiler, which starts at SYNENT ($A1C3), uses the pseudo-instructions in the Syntax Tables to verify the correctness of the source line and to generate the tokenized statements.
Chapter Five

Syntax Stack
The pre-compiler uses a LIFO stack in its processing. Each time a non-terminal vector ("GOSUB") is executed, the pre-compiler must remember where the call was made from. It must also remember the current locations in the input buffer (source statement) and the output buffer (tokenized statement) in case the called routine reports fail and backup is required. This LIFO stack is called the Syntax Stack.

The Syntax Stack starts at $480 at the label SIX. The stack is 256 bytes in size. Each entry in the stack is four bytes long. The stack can hold 64 levels of non-terminal calls. If a sixty-fifth stack entry is attempted, the LINE TOO LONG error is reported. (This error should be called LINE TOO COMPLEX, but the line is most likely too long also.)

The first byte of each stack entry is the current input index (CIX). The second byte is the current output index (COX). The final two bytes are the current address within the syntax tables.

The current stack level is managed by the STKLVL ($A9) cell. STKLVL maintains a value from $00 to $FC, which is the displacement to the current top of the stack entry.

Initialization
The editor has saved an address in SRCADR ($96). This address is the address, minus 1, of the current statement’s ABML instructions in the Syntax Tables. The current input index (CIX) and the current output index (COX) are also preset by the editor.

The initialization code resets the syntax stack manager (STKLVL) to zero and loads the first stack entry with the values in CIX, COX, and CPC — the current program counter, which holds the address of the next pseudo-instruction in the Syntax Tables.

PUSH
Values are placed on the stack by the PUSH routine ($A228). PUSH is entered with the new current pseudo-program counter value on the CPU stack. PUSH saves the current CIX, COX, and CPC on the syntax stack and increments STKLVL. Next, it sets a new CPC value from the data on the CPU stack. Finally, PUSH goes to NEXT.
Chapter Five

POP
Values are removed from the stack with the POP routine ($A252). POP is entered with the 6502 carry flag indicating pass/fail. If the carry is clear, then pass is indicated. If the carry is set, then fail is indicated.

POP first checks STKLVL. If the current value is zero, then the pre-compiler is done. In this case, POP returns to the editor via RTS. The carry bit status informs the editor of the pass/fail status.

If STKLVL is not zero, POP decrements STKLVL.

At this point, POP examines the carry bit status. If the carry is clear (pass), POP goes to NEXT. If the carry is set (fail), POP goes to FAIL.

NEXT and the Processes It Calls
After initialization is finished and after each Syntax Table instruction is processed, NEXT is entered to process the next syntax instruction.

NEXT starts by calling NXSC to increment CPC and get the next syntax instruction into the A register. The instruction value is then tested to determine which syntax instruction code it is and where to go to process it.

If the Syntax Instruction is OR ($02) or RTN ($03), then exit is via POP. When POP is called due to these two instructions, the carry bit is always clear, indicating pass.

ERNTV. If the instruction is RNTV (‘GOSUB’ $80 — $FF), then ERNTV ($A201) is entered. This code calculates the new CPC value, then exits via PUSH.

GETADR. If the instruction is ANTV ($00) or the deus ex machina ESRT ($01) instruction, then GETADR is called. GETADR obtains the following two-byte address from the Syntax Table.

If the instruction was ANTV, then GETADR exits via PUSH.

If the instruction was ESRT, then GETADR calls the external routine indicated. The external routine will report pass/fail via the carry bit. The pass/fail condition is examined at $A1F0. If pass is indicated, then NEXT is entered. If fail is indicated, then FAIL is entered.

TERMTST. If the instruction is VEXP ($0E), then the code at $A1F9 will go to TERMTST ($A2A9), which will cause the code
at $A2AF to be executed for VEXP. This code obtains the address, minus 1, of the ABML for the <expression> in the Syntax Table and exits via PUSH.

**ECHNG.** If the instruction was CHNG ($0F), then ECHNG ($A2BA) is entered via tests at $A1F9 and $A2AB. ECHNG will increment CPC and obtain the change-to token which will then replace the last previously generated token in OUTBUFF. ECHNG exits via RTS, which will take control back to NEXT.

**SRCONT.** The Operator Token Instructions ($10-$7F) are handled by the SRCONT routine. SRCONT is called via tests at $A1F9 and $A2AD. SRCONT will examine the current source symbol to see if it matches the symbol represented by the operator token. When SRCONT has made its determination, it will return to the code at $A1FC. This code will examine the pass/fail (carry clear/set) indicator returned by SRCONT and take the appropriate action. (The SRCONT routine is detailed on the next page.)

**FAIL**

If any routine returns a fail indicator, the FAIL code at $A26C will be entered. FAIL will sequentially examine the instructions, starting at the Syntax Table address pointed to by CPC, looking for an OR instruction.

If an OR instruction is found, the code at $A27D will be entered. This code first determines if the current statement symbol is the rightmost source symbol to be examined thus far. If it is, it will update MAXCIX. The editor will use MAXCIX to set the inverse video flag if the statement is erroneous. Second, the code restores CIX and COX to their before-failure values and goes to NEXT to try this new OR choice.

If, while searching for an OR instruction, FAIL finds a RTN instruction, it will call POP with the carry set. Since the carry is set, POP will re-enter FAIL once it has restored things to the previous calling level.

All instruction codes other than OR and RTN are skipped over by FAIL.
Pre-compiler Subroutine Descriptions

**SRCONT ($A2E6)**
The SRCONT code will be entered when an operator token instruction is found in the Syntax Tables by the main pre-compiler code. The purpose of the routine is to determine if the current source symbol in the user’s line matches the terminal symbol represented by the operator token. If the symbols match, the token is placed into the output buffer and *pass* is returned. If the symbols do not match, *fail* is returned.

SRCONT uses the value of the operator token to access the terminal symbol name in the Operator Name Table. The characters in the source symbol are compared to the characters in the terminal symbol. If all the characters match, *pass* is indicated.

**TNVAR, TSVAR ($A32A)**
These *deus ex machina* routines are called by the ESRT instruction. The purpose of the routines is to determine if the current source symbol is a valid numeric (TNVAR) or string (TSVAR) variable. If the source symbol is not a valid variable, *fail* is returned.

When *pass* is indicated, the routine will put a variable token into the output buffer. The variable token ($80-$FF) is an index into the Variable Name Table and the Variable Value Table, plus $80.

The Variable Name Table is searched. If the variable is already in the table, the token value for the existing variable is used. If the variable is not in the table, it will be inserted into both tables and a new token value will be used.

A source symbol is considered a valid variable if it starts with an alphabetic character and it is not a symbol in the Operator Name Table, which includes all the reserved words.

The variable is considered to be a string if it ends with $; otherwise it is a numeric variable. If it is a string variable, $ is stored with the variable name characters.

The routine also determines if the variable is an array by looking for (. If the variable is an array, ( is stored with the variable name characters in the Variable Name Table. As a result, ABC, ABC$, and ABC(n) are all recognized as different variables.
TNCON ($A400)
TNCON is called by the ESRT instruction. Its purpose is to examine the current source symbol for a numeric constant, using the floating point package. If the symbol is not a numeric constant, the routine returns fail.

If the symbol is a numeric constant, the floating point package has converted it to a floating point number. The resulting six-byte constant is placed in the output buffer preceded by the $0E numeric constant token. The routine then exits with pass indicated.

TSCON ($A428)
TSCON is called by the ESRT instruction. Its purpose is to examine the current symbol for a string constant. If the symbol is not a string constant, the routine returns fail.

If the first character of the symbol is ‘‘, the symbol is a string constant. The routine will place the string constant token ($0F) into the output buffer, followed by a string length byte, followed by the string characters.

The string constant consists of all the characters that follow the starting double quote up to the ending double quote. If the EOL character ($9B) is found before the ending double quote, an ending double quote is assumed. The EOL is not part of the string. The starting and ending double quotes are not saved with the string. All 256 character codes except $9B (EOL) and $22 (‘’) are allowed in the string.

SEARCH ($A462)
This is a general purpose table search routine used to find a source symbol character string in a table.

The table to be searched is assumed to have entries which consist of a fixed length part (0 to 255 bytes) followed by a variable length ATASCII part. The last character of the ATASCII part is assumed to have the most significant bit ($80) on. The last table entry is assumed to have the first ATASCII character as $00.

Upon entry, the X register contains the length of the fixed part of the table (0 to 255). The A, Y register pair points to the start of the table to be searched. The source string for comparison is pointed to by INBUFF plus the value in CIX.

Upon exit, the 6502 carry flag is clear if a match was found, and set if no match was found. The X register points to the end
of the symbol, plus 1, in the buffer. The SRCADR ($95) two-byte cell points to the matched table entry. STENUM ($AF) contains the number, relative to zero, of the matched table entry.

**SETCODE (A2C8)**
The SETCODE routine is used to place a token in the next available position in the output (token) buffer. The value in COX determines the current displacement into the token buffer. After the token is placed in the buffer, COX is incremented by one. If COX exceeds 255, the LINE TOO LONG error message is generated.
Execution Overview

During the editing and pre-compiling phase, the user’s statements were checked for correct syntax, tokenized, and put into the Statement Table. Then direct statements were passed to the Program Executor for immediate processing, while program statements awaited later processing by the Program Executor.

We now enter the execution phase of Atari BASIC. The Program Executor consists of three parts: routines which simulate the function of individual statement types; an expression execution routine which processes expressions (for example, A + B + 3, A$(1,3), "HELP", A(3)+7.26E-13); and the Execution Control routine, which manages the whole process.

Execution Control

Execution Control is invoked in two situations. If the user has entered a direct statement, Execution Control does some initial processing and then calls the appropriate statement execution routine to simulate the requested operation. If the user has entered RUN as a direct statement, the statement execution routine for RUN instructs Execution Control to start processing statements from the beginning of the statement table.

When the editor has finished processing a direct statement, it initiates the Execution Control routine EXECNL ($A95F). Execution Control’s job is to manage the process of statement simulation.

The editor has saved the address of the statement it processed in STMCUR and has put the statement in the Statement Table. Since this is a direct statement, the line number is $8000, and the statement is saved as the last line in the Statement Table.

The fact that a direct statement is always the last statement in the Statement Table gives a test for the end of a user’s program.

The high-order byte of the direct statement line number ($8000) has its most significant bit on. Loading this byte ($80)
into the 6502 accumulator will set the minus flag on. The line number of any program statement is less than or equal to $7FFF. Loading the high order byte ($7F or less) of a program line number into the accumulator will set the 6502 minus flag off. This gives a simple test for a direct statement.

**Initialization**

Execution Control uses several parameters to help it manage the task of statement execution.

- STMCUR holds the address in the Statement Table of the line currently being processed.
- LLNGTH holds the length of the current line.
- NXTSTD holds the displacement in the current line of the next statement to process.

STMCUR already contains the correct value when Execution Control begins processing. SETLN1 ($B81B) is called to store the correct values into LLNGTH and NXTSTD.

**Statement Execution**

Since the user may have changed his or her mind about execution, the routine checks to see if the user hit the break key. If the user did hit BREAK, Execution Control carries out XSTOP ($B793), the same routine that is executed when the STOP statement is encountered. At the end of its execution, the XSTOP routine gives control to the beginning of the editor.

If the user did not hit BREAK, Execution Control checks to see whether we are at the end of the tokenized line. Since this is the first statement in the line, we can’t be at the end of the line. So why do the test? Because this part of the routine is executed once for each statement in the line in order to tell us when we do reach the end of the line. (The end-of-line procedure will be discussed later in this chapter.)

The statement length byte (the displacement to the next statement in the line) is the first byte in a statement. (See Chapter 3.) The displacement to this byte was saved in NXTSTD. Execution Control now loads this new statement’s displacement using the value in NXTSTD.

The byte after the statement length in the line is the statement name token. Execution Control loads the statement name token into the A register. It saves the displacement to the next byte, the first of the statement’s tokens, in STINDEX for the use of the statement simulation routines.
Chapter Six

The statement name token is used as an index to find this statement’s entry in the Statement Execution Table. Each table entry consists of the address, minus 1, of the routine that will simulate that statement. This simulation routine is called by pushing the address from the table onto the 6502 CPU stack and doing an RTS. Later, when a simulation routine is finished, it can do an RTS and return to Execution Control. (The name of most of the statement simulation routines in the BASIC listing is the statement name preceded by an X: XFOR, XRUN, XLIST.)

Most of the statement simulation routines return to Execution Control after processing.

Execution Control again tests for BREAK and checks for the end of the line. As long as we are not at end-of-line, it continues to execute statements. When we reach end-of-line, it does some end-of-line processing.

End-of-line Handling in a Direct Statement
When we come to the end of the line in a direct statement, Execution Control has done its job and jumps to SNX3. The READY message is printed and control goes back to the Program Editor.

End-of-line Handling during Program Execution
Program execution is initiated when the user types RUN. Execution Control handles RUN like any other direct statement. The statement simulation routine for RUN initialises STMCUR, NXTSTD, and LLNGTH to indicate the first statement of the first line in the Statement Table, then returns to Execution Control. Execution Control treats this first program statement as the next statement to be executed, picking up the statement name tokens and calling the simulation routines.

Usually, Execution Control is unaware of whether it is processing a direct statement or a program statement. End-of-line is the only time the routine needs to make a distinction.

At the end of every program line, Execution Control gets the length of the current line and calls GNXTL to update the address in STMCUR to make the next line in the Statement Table the new current line. Then it calls TENDST ($A9E2) to test the new line number to see if it is another program line or a direct statement. If it is a direct statement, we are at the end of the user’s program.
Chapter Six

Since the direct statement includes the RUN command that started program execution, Execution Control does not execute the line. Instead, Execution Control calls the same routine that would have been called if the program had contained an END statement (XEND, at $B78D). XEND does some end-of-program processing, causes READY to be printed, and returns to the beginning of the editor.

If we are not at the end of the user’s program, processing continues with the new current line.

Execution Control Subroutines

TENDST ($A9E2)
Exit parameters: The minus flag is set on if we are at the end of program.

This routine checks for the end of the user’s program in the Statement Table.

The very last entry in the Statement Table is always a direct statement. Whenever the statement indicated by STMCUR is the direct statement, we have finished processing the user’s program.

The line number of a direct statement is $8000. The line number of any other statement is $7FFF or less. TENDST determines if the current statement is the direct statement by loading the high-order byte of the line number into the A register. This byte is at a displacement of one from the address in STMCUR. If this byte is $80 (a direct statement), loading it turns the 6502 minus flag on. Otherwise, the minus flag is turned off.

GETSTMT ($A9A2)
Entry parameters: TSLNUM contains the line number of the statement whose address is required.

Exit parameters: If the line number is found, the STMCUR contains the address of the statement and the carry flag is set off (clear). If the line number does not exist, STMCUR contains the address where a statement with that line number should be, and the carry flag is set on (set).

The purpose of this routine is to find the address of the statement whose line number is contained in TSLNUM.

The routine saves the address currently in STMCUR into SAVCUR and then sets STMCUR to indicate the top of the
Chapter Six

Statement Table. The line whose address is in STMCUR is called the current line or statement.

GETSTMT then searches the Statement Table for the statement whose line number is in TSLNUM. The line number in TSLNUM is compared to the line number of the current line. If they are equal, then the required statement has been found. Its address is in STMCUR, so GETSTMT clears the 6502 carry flag and is finished.

If TSLNUM is smaller than the current statement line number, GETSTMT gets the length of the current statement by executing GETLL ($A9DD). GNXTL ($A9D0) is executed to make the next line in the statement table the current statement by putting its address into STMCUR. GETSTMT then repeats the comparison of TSLNUM and the line number of the current line in the same manner.

If TSLNUM is greater than the current line number, then a line with this line number does not exist. STMCUR already points to where the line should be, the 6502 carry flag is already set, and the routine is done.

GETLL ($A9DD)
Entry parameters: STMCUR indicates the line whose length is desired.

Exit parameters: Register A contains the length of the current line.

GETLL gets the length of the current line (that is, the line whose address is in STMCUR).

The line length is at a displacement of two into the line. GETLL loads the length into the A register and is done.

GNXTL ($A9D0)
Entry parameters: STMCUR contains the address of the current line, and register A contains the length of the current line.

Exit parameters: STMCUR contains the address of the next line.

This routine gets the next line in the statement table and makes it the current line.

GNXTL adds the length of the current line (contained in the A register) to the address of the current line in STMCUR. This process yields the address of the next line in the statement table, which replaces the value in STMCUR.
SETLN1 ($B81B)
Entry parameters: STMCUR contains the address of the current line.
Exit parameters: LLNGTH contains the length of the current line. NXTSTD contains the displacement in the line to the next statement to be executed (in this case, the first statement in the line).
This routine initializes several line parameters so that Execution Control can process the line.
The routine gets the length of the line, which is at a displacement of two from the start of the line.
SETLN1 loads a value of three into the Y register to indicate the displacement into the line of the first statement and stores the value into NXTSTD as the displacement to the next statement for execution.

SETLINE ($B818)
Entry parameters: TSLNUM contains the line number of a statement.
Exit parameters: STMCUR contains the address of the statement whose line number is in TSLNUM. LLNGTH contains the length of the line. NXTSTD contains the displacement in the line to the next statement to be executed (in this case, the first statement in the line). Carry is set if the line number does not exist.
This routine initializes several line parameters so that execution control can process the line.
SETLINE first calls GETSTMT ($A9A2) to find the address of the line whose number is in TSLNUM and put that address into STMCUR. It then continues exactly like SETLN1.
Chapter Seven

Execute Expression

The Execute Expression routine is entered when the Program Executor needs to evaluate a BASIC expression within a statement. It is also the executor for the LET and implied LET statements.

Expression operators have an order of precedence; some must be simulated before others. To properly evaluate an expression, Execute Expression rearranges it during the evaluation.

Expression Rearrangement Concepts

Operator precedence rules in algebraic expressions are so simple and so unconscious that most people aren't aware of following them. When you evaluate a simple expression like \( Y = AX^2 + BX + C \), you don't think: "Exponentiation has a higher precedence than multiplication, which has a higher precedence than addition; therefore, I will first square the \( X \), then perform the multiplication." You just do it.

Computers don't develop habits or common sense — they have to be specifically commanded. It would be nice if we could just type \( Y = AX^2 + BX + C \) into our machine and have the computer understand, but instead we must separate all our variables with operators. We also have to learn a few new operators, such as * for multiply and ^ for exponentiation.

Given that we are willing to adjust our thinking this much, we enter \( Y = A \times X^2 + B \times X + C \). The new form of expression does not quite have the same feel as \( Y = AX^2 + BX + C \); we have translated normal human patterns halfway into a form the computer can use.

Even the operation \( X^2 \) causes another problem for the computer. It would really prefer that we give it the two values first, then tell it what to do with them. Since the computer still needs separators between items, we should write \( X^2 \) as \( X, 2, ^ \).

Now we have something the computer can work with. It can obtain the two values \( X, 2 \), apply the operator ^, and get a result without having to look ahead.
Chapter Seven

If we were to transcribe $X^2A$ in the same manner, we would have $X,2,^2,A,*$. The value returned by $X,2,^2$ is the first value to multiply, so the value pair for multiplication is $(X,2,^2)$ and $A$. Again we have two values followed by an operator, and the computer can understand.

If we continue to transcribe the expression by pairing values and operators, we find that we don't want to add the value $X^2A$ to $B$; we want to add the value $X^2A$ to $BX$. Therefore, we need to tell the computer $X,2,^2,A,*$, $B,X,*$, $+$. The value pair for the operator $+$ is $(X,2,^2,A,*$) and $(B,X,*$).

The value pair for the final operation, $=$, is $(X,2,^2,A,*$, $B,X,*$, $+,$ $C,$ $+,$ $Y,$ $=,$) and $Y$. So the complete translation of $Y = AX^2 + BX + C$ is $X,2,^2,A,*$, $B,X,*$, $+,$ $C,$ $+,$ $Y,$ $=,$.

Very few people other than Forth programmers put up with this form of expression transcription. Therefore, Atari BASIC was designed to perform this translation for us, provided we use the correct symbols, like * and $^2$.

The Expression Rearrangement Algorithm
The algorithm for expression rearrangement requires two LIFO stacks for temporary storage of the rearranged terms. The Operator Stack is used for temporarily saving operators; the Argument Stack is used for saving arguments. Arguments are values consisting of variables, constants, and the constant-like values resulting from previous expression operations.

Operator Precedence Table
The Atari BASIC User's Manual lists the operators by precedence. The highest-precedence operators, like $<$, $>$, and $=$, are at the top of the list; the lowest-precedence operator, OR, is at the bottom. The operators at the top of the list get executed before the operators at the bottom of the list.

The operators in the precedence table are arranged in the same order as the Operator Name Table. Thus the token values can be used as direct indices to obtain an operator precedence value.

The entry for each operator in the Operator Precedence Table contains two precedence values, the go-onto-stack precedence and the come-off-stack precedence. When a new operator has been plucked from an expression, its go-onto-stack precedence is tested in relation to the top-of-stack operator's come-off-stack precedence.
Expression Rearrangement Procedure

The symbols of the expression (the arguments and the operators) are accessed sequentially from left to right, then rearranged into their correct order of precedence by the following procedure:

1. Initialize the Operator Stack with the Start Of Expression (SOE) operator.
2. Get the next symbol from the expression.
3. If the symbol is an argument (variable or constant), place the argument on the top of the Argument Stack. Go to step 2.
4. If the symbol is an operator, save the operator in the temporary save cell, SAVEOP.
5. Compare the go-onto-stack precedence of the operator in SAVEOP to the come-off stack precedence of the operator on the top of the Operator Stack.
6. If the top-of-stack operator’s precedence is less than the precedence of the SAVEOP operator, then the SAVEOP operator is pushed onto the Operator Stack. When the push is done, go back to step 2.
7. If the top-of-stack operator’s precedence is equal to or greater than the precedence of the SAVEOP operator, then pop the top-of-stack operator and execute it. When the execution is done, go back to step 5 and continue.

The Expression Rearrangement Procedure has one apparent problem. It seems that there is no way to stop it. There are no exits for the "evaluation done" condition. This problem is handled by enclosing the expression with two special operators: the Start Of Expression (SOE) operator, and the End Of Expression (EOE) operator. Remember that SOE was the first operator placed on the Operator Stack, in step 1. Execution code for the SOE operator will cause the procedure to be exited in step 7, when SOE is popped and executed. The EOE operator is never executed. EOE’s function is to force the execution of SOE.

The precedence values of SOE and EOE are set to insure that SOE is executed only when the expression evaluation is finished. The SOE come-off-stack precedence is set so that its value is always less than all the other operators’ go-onto-stack precedence values. The EOE go-onto-stack precedence is set so that its value is always equal to or less than all the other
operators’ (including SOE’s) come-off-stack precedence values.

Because SOE and EOE precedence are set this way, no operator other than EOE can cause SOE to be popped and executed. Second, EOE will cause all stacked operators, including SOE, to be popped and executed. Since SOE is always at the start of the expression and EOE is always at the end of the expression, SOE will not be executed until the expression is fully evaluated.

In actual practice, the SOE operator is not physically part of the expression in the Statement Table. The Expression Rearrangement Procedure initializes the Operator Stack with the SOE operator before it begins to examine the expression.

There is no single operator defined as the End Of Expression (EOE) operator. Every BASIC expression is followed by a symbol like :, THEN, or the EOL character. All of these symbols function as operators with precedence equivalent to the precedence of our phantom EOE operator. The THEN token, for example, serves a dual purpose. It not only indicates the THEN action, but also acts as the EOE operator when it follows an expression.

Expression Rearrangement Example
To illustrate how the expression evaluation procedure works, including expression rearrangement, we will evaluate our \( Y = A \times X^{2} + B \times X + C \) example and see how the expression is rearranged to \( X,2,\wedge,A,\times,B,\times,+,C,+,Y,= \) with a correct result. To work our example, we need to establish a precedence table for the operators. The values in Figure 7-1 are similar to the actual values of these operators in Atari BASIC. The lowest precedence value is zero; the highest precedence value is $0F.$

**Figure 7-1. Example Precedence Table**

<table>
<thead>
<tr>
<th>operator symbol</th>
<th>go-on-stack precedence</th>
<th>come-off-stack precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOE</td>
<td>NA</td>
<td>$00</td>
</tr>
<tr>
<td>+</td>
<td>$09</td>
<td>$09</td>
</tr>
<tr>
<td>*</td>
<td>$0A</td>
<td>$0A</td>
</tr>
<tr>
<td>( \wedge )</td>
<td>$0C</td>
<td>$0C</td>
</tr>
<tr>
<td>=</td>
<td>$0F</td>
<td>$01</td>
</tr>
<tr>
<td>( ! ) (EOE)</td>
<td>$00</td>
<td>NA</td>
</tr>
</tbody>
</table>
Symbol values and notations. In the example steps, the term PS\textsubscript{n} refers to step \textit{n} in the Expression Rearrangement Procedure (page 57). Step 5, for instance, will be called PS5.

In the actual expression, the current symbol will be underlined. If \( B \) is the current symbol, then the actual expression will appear as \( Y = A^2 X + B^2 X + C \). In the rearranged expression, the symbols which have been evaluated up to that point will also be underlined.

The values of the variables are:

\[
\begin{align*}
A &= 2 \\
B &= 4 \\
C &= 6 \\
X &= 3
\end{align*}
\]

The variable values are assumed to be accessed when the variable arguments are popped for operator execution.

The end-of-expression operator is represented by \( ! \).

Example step 1.

Actual Expression: \( Y = A^2 X + B^2 X + C! \)

Rearranged Expression: \( X, 2, ^2, A, ^2, B, ^2, X, +, C, +, Y, =, ! \)

Argument Stack: \( Y \)

Operator Stack: \( \text{SOE} \)

SAVEOP:

PS1 has been executed. The Operator Stack has been initialized with the SOE operator. We are ready to start processing the expression symbols.

Example step 2.

Actual Expression: \( Y = A^2 X + B^2 X + C! \)

Rearranged Expression: \( X, 2, ^2, A, ^2, B, ^2, X, +, C, +, Y, =, ! \)

Argument Stack: \( Y \)

Operator Stack: \( \text{SOE} \)

SAVEOP:

The first symbol, \( Y \), has been obtained and stacked in the Argument Stack according to PS2 and PS3.

Example step 3.

Actual Expression: \( Y = A^2 X + B^2 X + C! \)

Rearranged Expression: \( X, 2, ^2, A, ^2, B, ^2, X, +, C, +, Y, =, ! \)

Argument Stack: \( Y \)

Operator Stack: \( \text{SOE}, = \)

SAVEOP: \( = \)
Chapter Seven

Operator = has been obtained via PS2. The relative precedences of SOE ($00) and = ($0F) dictate that the = be placed on the Operator Stack via PS6.

Example step 4.

Actual Expression: $Y = A_X^2 + B_X + C!$
Argument Stack: $Y, A$
Operator Stack: SOE, =
SAVEOP:

The next symbol is A. This symbol is pushed onto the Argument Stack via PS3.

Example step 5.

Actual Expression: $Y = A X^2 + B X + C!$
Argument Stack: $Y, A$
Operator Stack: SOE, =, *
SAVEOP:

The next symbol is the operator *. The relative precedence of * and = dictates that * be pushed onto the Operator Stack.

Example step 6.

Actual Expression: $Y = A _X^2 + B X + C!$
Argument Stack: $Y, A, X$
Operator Stack: SOE, =, *
SAVEOP:

The next symbol is the variable X. This symbol is stacked on the Argument Stack according to PS3.

Example step 7.

Actual Expression: $Y = A X_\wedge 2 + B X + C!$
Argument Stack: $Y, A, X$
Operator Stack: SOE, =, *, \wedge
SAVEOP: \wedge

The next symbol is \wedge. The relative precedence of the and the * dictate that \wedge be stacked via PS6.
Example step 8.

Actual Expression:  \( Y = A \times X^2 + B \times X + C \)
Rearranged Expression:  \( X, 2, ^, A, *, B, X, *, +, C, +, Y, =, ! \)
Argument Stack:  \( Y, A, X, 2 \)
Operator Stack:  \( \text{SOE}, =, *, ^ \)
SAVEOP:

The next symbol is 2. This symbol is stacked on the Argument Stack via PS3.

Example step 9.

Actual Expression:  \( Y = A \times X^2 + B \times X + C \)
Rearranged Expression:  \( X, 2, ^, A, *, B, X, *, +, C, +, Y, =, ! \)
Argument Stack:  \( Y, A, 9 \)
Operator Stack:  \( \text{SOE}, =, * \)
SAVEOP: +

The next symbol is the operator +. The precedence of the operator that was at the top of the stack, \(^\), is greater than the precedence of +. PS7 dictates that the top-of-stack operator be popped and executed.

The \(^\) operator is popped. Its execution causes arguments X and 2 to be popped from the Argument Stack, replacing the variable with the value that it represents and operating on the two values yielded: \( X^2 = 3^2 = 9 \). The resulting value, 9, is pushed onto the Argument Stack. The + operator remains in SAVEOP. We continue at PS5.

Note that in the rearranged expression the first symbols, \( X, 2, ^ \), have been evaluated according to plan.

Example step 10.

Actual Expression:  \( Y = A \times X^2 + B \times X + C \)
Rearranged Expression:  \( X, 2, ^, A, *, B, X, *, +, C, +, Y, =, ! \)
Argument Stack:  \( Y, 18 \)
Operator Stack:  \( \text{SOE}, = \)
SAVEOP: +

This step originates at PS5. The SAVEOP operator, +, has a precedence that is less than the operator which was at the top of the stack, *. Therefore, according to PS7, the * is popped and executed.

The execution of * results in \( A \times 9 = 2 \times 9 = 18 \). The resulting value is pushed onto the Argument Stack.
Example step 11.

Actual Expression: \( Y = A \times X^2 \pm B \times X + C! \)
Argument Stack: \( Y, 18 \)
Operator Stack: \( \text{SOE}, =, + \)
SAVEOP:

When step 10 finished, we went to PS5. The operator in SAVEOP was +. Since + has a higher precedence than the top-of-stack operator, =, the + operator was pushed onto the Operator Stack via PS6.

Example step 12.

Actual Expression: \( Y = A \times X^2 + B \times X + C! \)
Argument Stack: \( Y, 18, B \)
Operator Stack: \( \text{SOE}, =, + \)
SAVEOP:

The next symbol is the variable B, which is pushed onto the Argument Stack via PS3.

Example step 13.

Actual Expression: \( Y = A \times X^2 + B \times X + C! \)
Argument Stack: \( Y, 18, B \)
Operator Stack: \( \text{SOE}, =, +, * \)
SAVEOP: *

The next symbol is the operator *. Since * has a higher precedence than the top-of-stack +, * is pushed onto the stack via PS6.

Example step 14.

Actual Expression: \( Y = A \times X^2 + B \times X + C! \)
Argument Stack: \( Y, 18, B, X \)
Operator Stack: \( \text{SOE}, =, +, * \)
SAVEOP:

The variable X is pushed onto the Argument Stack via PS3.

Example step 15.

Actual Expression: \( Y = A \times X^2 + B \times X + C! \)
Rearranged Expression: \( X, 2, A, *, B, X, *, +, C, +, Y, =, ! \)
Argument Stack: Y,18,12
Operator Stack: SOE, =, +
SAVEOP: +

The operator + is retrieved from the expression. Since + has a lower precedence than * which is at the top of the stack, * is popped and executed.
The execution of * causes B*X = 4*3 = 12. The resulting value of 12 is pushed onto the Argument Stack. We will continue at PS5 via the PS7 exit rule.

Example step 16.

Actual Expression: Y = A*X\^2 + B*X + C!
Rearranged Expression: X,2,^,A,*,B,X,*,+,C,+,Y,=,!
Argument Stack: Y,30
Operator Stack: SOE,=,+
SAVEOP:

This step starts at PS5. The SAVEOP operator, +, has precedence that is equal to the precedence of the top-of-stack operator, also +. Therefore, + is pushed from the operator stack and executed. The results of the execution cause 18+12, or 30, to be pushed onto the Argument Stack. PS5 is called.

Example step 17.

Actual Expression: Y = A*X\^2 + B*X + C!
Rearranged Expression: X,2,^,A,*,B,X,*,+,C,+,Y,=,!
Argument Stack: Y,30
Operator Stack: SOE,=,+
SAVEOP:

This step starts at PS5. The SAVEOP is +. The top-of-stack operator, =, has a lower precedence than +; therefore, + is pushed onto the stack via PS6.

Example step 18.

Actual Expression: Y = A*X\^2 + B*X + C!
Rearranged Expression: X,2,^,A,*,B,X,*,+,C,+,Y,=,!
Argument Stack: Y,30,C
Operator Stack: SOE,=,+
SAVEOP:

The variable C is pushed onto the Argument Stack via PS3.
Chapter Seven

Example step 19.

Actual Expression: \( Y = A \times X^2 + B \times X + C! \)

Rearranged Expression: \( X,2,^,A,*,B,X,*,+,C,+,Y,=,! \)

Argument Stack: \( Y,36 \)

Operator Stack: \( \text{SOE},= \)

SAVEOP: !

The EOE operator ! is plucked from the expression. The EOE has a lower precedence than the top-of-stack + operator. Therefore, + is popped and executed. The resulting value of \( 30 + 6, 36 \), is pushed onto the Argument Stack. PS5 will execute next.

Example step 20.

Actual Expression: \( Y = A \times X^2 + B \times X + C! \)

Rearranged Expression: \( X,2,^,A,*,B,X,*,+,C,+,Y,=,! \)

Argument Stack: 

Operator Stack: \( \text{SOE} \)

SAVEOP: !

This step starts at PS5. The ! operator has a lower precedence than the top-of-stack = operator, which is popped and executed. The execution of = causes the value 36 to be assigned to \( Y \). This leaves the Argument Stack empty. PS5 will be executed next.

Example step 21.

Actual Expression: \( Y = A \times X^2 + B \times X + C! \)

Rearranged Expression: \( X,2,^,A,*,B,X,*,+,C,+,Y,=,! \)

Argument Stack: 

Operator Stack: 

SAVEOP: !

The ! operator in SAVEOP causes the SOE operator to be popped and executed. The execution of SOE terminates the expression evaluation.

Note that the rearranged expression was executed exactly as predicted.

Mainline Code

The Execute Expression code implements the Expression Rearrangement Procedure. The mainline code starts at the EXEXPR label at $AAE0. The input to EXEXPR starts at the current token in the current statement. STMCUR points to the
current statement. STINDEX contains the displacement to the
current token in the STM CUR statement. The output of
EXEXPR is whatever values remain on the top of the argument
stack when the expression evaluation is finished.

In the following discussion, PSn refers to the procedure
step n in the Expression Rearrangement Procedure.

PS1, initialization, occurs when EXEXPR is entered.
EXPINT is called to initialize the operator and argument stacks.
EXPINT places the SOE operator on the operator stack.

PS2, which obtains the next token, directly follows
initialization at EXNXT ($AAE3). The code calls EGTK O N K E N T
to get the next expression symbol and classify it. If the token is an
argument, the carry will be set. If the token is an operator, the
carry will be clear.

If the token is an argument, PS3 is implemented via a call to
ARGPUSH. After the argument is pushed onto the argument
stack, EXNXT (PS2) will receive control.

If the token was an operator, then the code at EXOT
($AAEE) will be executed. This code implements PS4 by saving
the token in EXSVOP.

PS5, which compares the precedents of the EXSVOP token
and the top-of-stack token, follows EXOT at EXPTST ($AAFA).
This code also executes the SOE operator. If SOE is popped,
then Execute Expression finishes via RTS.

If the top-of-stack operator precedence is less than the
EXSVOP operator precedence, PS6 is implemented at
EOPUSH ($AB15). EOPUSH pushes EXSVOP onto the
operator stack and then goes to EXNXT (PS2).

If the top-of-stack operator precedence is greater than or
equal to the EXSVOP operator precedence, then PS7 is
implemented at EXOPOP ($AB0B). EXOPOP will pop the top-
of-stack operator and execute it by calling EXOP. When EXOP
is done, control passes to EXPTST (PS5).

Expression Evaluation Stacks
The two expression evaluation stacks, the Argument Stack and
the Operator Stack, share a single 256-byte memory area. The
Argument Stack grows upward from the lower end of the 256-
byte area. The Operator Stack grows downward from the
upper end of the 256-byte area.

The 256-byte stack area is the multipurpose buffer at the
start of the RAM tables. The buffer is pointed to by the
ARGSTK (also ARGOPS) zero-page pointer at $80. The current index into the Argument Stack is maintained by ARSLVL ($AA). When the Argument Stack is empty, ARSLVL is zero.

The OPSTKX cell maintains the current index into the Operator Stack. When the Operator Stack is initialized with the SOE operator, OPSTKX is initialized to $FF. As operators are added to the Operator Stack, OPSTKX is decremented. As arguments are added to the Argument Stack, ARSLVL is incremented.

Since the two stacks share a single 256-byte memory area, there is a possibility that the stacks will run into each other. The code at $ABC1 is used to detect a stack collision. It does this by comparing the values in ARSLVL and OPSTKX. If ARSLVL is greater than or equal to OPSTKX, then a stack collision occurs, sending the STACK OVERFLOW error to the user.

Operator Stack
Each entry on the Operator Stack is a single-byte operator-type token. Operators are pushed onto the stack at EXOPUSH ($AB15) and are popped from the stack at EXOPOP ($AB0B).

Argument Stack
Each entry on the Argument Stack is eight bytes long. The format of these entries is described in Figures 7-2, 7-3, and 7-4, and are the same as the formats for entries in the Variable Value Table.

Unlike the Variable Value Table, the Argument Stack must deal with both variables and constants. In Figure 7-2, we see that VNUM is used to distinguish variable entries from constant entries.

The SADR and AADR fields in the entries for strings and arrays are of special interest. (See Figures 7-3 and 7-4.) When a string or array variable is dimensioned, space for the variable is created in the string/array space. The displacement to the start of the variable’s area within the string/array space is placed in the SADR/AADR fields at that time. A displacement is used rather than an absolute address because the absolute address can change if any program changes are made after the DIM statement is executed.

Execute Expression needs these values to be absolute address values within the 6502 address space. When a string/array variable is retrieved from the Variable Value Table,
the displacement is transformed to an absolute address. When (and if) the variable is put back into the Variable Value Table, the absolute address is converted back to a displacement.

The entries for string constants also deserve some special attention. String constants are the quoted strings within the user program. These strings become part of the tokenized statements in the Statement Table. When Execute Expression gets a string token, it will create a string constant Argument Stack entry. This entry’s SADR is an absolute address pointer to the string in the Statement Table. SLEN and SDIM are set to the actual length of the quoted string.

**Argument Work Area**

An argument which is currently being examined by Execute Expression is kept in a special zero-page Argument Work Area (AWA). The AWA starts at the label VTYPE at $D2.

**Figure 7-2. Argument Stack Entry**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>VTYPE</td>
<td>VNUM</td>
<td>DATA</td>
<td></td>
</tr>
</tbody>
</table>

- **Data Field.** Format depends on VTYPE.
- If VNUM = 0, the entry is a constant.
- If VNUM > 0, the entry is a variable. In this case, the VNUM value is the entry number in the Variable Value Table. The token representing this variable is VNUM+$80.

- $00 = Data is a six-byte floating point constant.
- $80 = Data represents an undimensioned string.
- $81 = Data represents a dimensioned string with a relative address pointer.
- $83 = Data represents a dimensioned string with an absolute address pointer.
- $40 = Data represents an undimensioned array.
- $41 = Data represents a dimensioned array with a relative address pointer.
- $43 = Data represents a dimensioned array with an absolute address pointer.
Chapter Seven

Figure 7-3. Argument Stack String Entry

0 1 2 4 6 8
VTYPE | VNUM | SADR | SLEN | SDIM

- Dimensioned length of the string. Valid only if `VTYPE = $81` or `$83`.
- Current length of the string. Valid only if `VTYPE = $81` or `$83`.
- String address. Valid only if `VTYPE = $81` or `$83`.
  - If `VTYPE = $81`, then `SADR` is the displacement of the start of the string from the start of the string/array space.
  - If `VTYPE = $83`, then `SADR` is the absolute address of the start of the string.

Figure 7-4. Argument Stack Array Entry

0 1 2 4 6 8
VTYPE | VNUM | AADR | DIM1 | DIM2

- When an array has been dimensioned as `A(D1,D2)`, this field contains the `D2` value. If an array was dimensioned as `A(D1)`, then this field is zero. The field is valid only if `VTYPE = $41` or `$43`.
- When an array has been dimensioned, as `A(D1,D2)` or as `A(D1)`, this field contains the `D1` value. The field is valid only if `VTYPE = $41` or `$43`.
- Array Address. Valid only if `VTYPE = $41` or `$43`.
  - If `VTYPE = $41`, the `AADR` is the displacement to the start of the array in the string/array space.
  - If `VTYPE = $43`, the `AADR` is the absolute address of the start of the string.
Chapter Seven

Operator Executions
An operator is executed when it is popped from the Operator Stack. Execute Expression calls EXOP at $AB20 to start this execution. The EXOP routine uses the operator token value as an index into the Operator Execution Table ($AA70). The operator execution address from this table, minus 1, is placed on the 6502 CPU stack. An RTS is then executed to begin executing the operator's code.

The names of the operator execution routines all begin with the characters XP.

All the Atari BASIC functions, such as PEEK, RND, and ABS, are executed as operators.

Most routines for the execution of the operators are very simple and straightforward. For example, the * operator routine, XPMUL ($AC96), pops two arguments, multiplies them via the floating point package, pushes the result onto the argument stack, and returns.

String, Array, DIM, and Function Operations
Any array reference in an expression may be found in one of two forms: A(x) or A(x,y). The indices x and y may be any valid expression. The intent of the indices is to reference a specific array element.

Before the specific element reference can take place, the x and/or y index expressions must be fully evaluated. To do this, the characters '(', ' and ') are made operators. The precedence of these operators forces things to happen in the correct sequence. Figure 7-5 shows the relative precedence of these operators for an array.

Figure 7-5. Array Operator Precedence

<table>
<thead>
<tr>
<th>operator symbol</th>
<th>go-on-stack precedence</th>
<th>come-off-stack precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>$0F</td>
<td>$02</td>
</tr>
<tr>
<td>, (comma)</td>
<td>$04</td>
<td>$03</td>
</tr>
<tr>
<td>)</td>
<td>$04</td>
<td>$0E</td>
</tr>
</tbody>
</table>

As a result of these precedence values, ( has a high enough precedence to go onto the stack, no matter what other operator is on the top of the stack.
The comma's go-on-stack precedence will force all operators except ( to be popped and executed. As a result, the x index sub-expression, in the expression $A(x, y)$, will be fully evaluated and the final $x$ index value will be pushed onto the Argument Stack.

The comma will then be placed onto the Operator Stack. Its come-off-stack precedence is such that no other operator, except ), will pop it off.

The ) operator precedence will force any $y$ index expression to be fully evaluated and the $y$ index result value to be placed onto the Argument Stack.

It will then force the comma operator to be popped and executed. This action results in a comma counter being incremented.

The ) will then force the ( to be popped and executed. The execution of ( results in the proper array element being referenced. The ( operator will pop the indices from the Argument Stack. The number of indices (either zero or one) to be popped is governed by the comma counter, which was incremented by one for each comma that was popped and executed.

Atari BASIC has numerous ( tokens, and each causes a different ( routine to be executed. These ( operators are array (CALPRN), string (CSLPRN), array DIM (CDLPRN), string DIM (CDSLPR), function (CFLPRN), and the expression grouping CLPRN operator. The Syntax Table pseudo-instruction CHNG is used to change the CLPRN token to the other ( tokens in accordance with the context of the grammar.

The expression operations for each of these various ( operators in relation to commas and ( is exactly the same. When ( is executed, the comma count will show how many arguments the operator's code must pop from the argument stack. Each of these arguments will have been evaluated down to a single value in the form of a constant.
Execution Boundary Conditions

BASIC Language statements can be divided into groups with related functions. The execution boundary statements, RUN, STOP, CONT and END, cause a BASIC program to start or stop executing. The routines which simulate these statements are XRUN, XSTOP, XCONT, and XEND.

**Program Termination Routines**
Any BASIC statement can be used as either a direct statement or a program statement, but some only make sense in one mode. The STOP statement has no real meaning when entered as a direct statement. When the statement simulation routine for STOP is asked to execute in direct mode, it does as little processing as possible and exits. Useful processing occurs only when STOP is a program statement.

**STOP ($B7A7).** The XSTOP and XEND routines are similar and perform some of the same tasks. The tasks common to both are handled by the STOP routine.

If this statement is not a direct statement, the STOP routine saves the line number of the current line in STOPLN. This line number is used later for printing the STOPed message. It is also used by the CONT simulation routine (XCONT) to determine where to restart program execution. (Since XEND also uses this routine, it is possible to CONTinue after an END statement in the middle of a program.)

The STOP routine also resets the LIST and ENTER devices to the screen and the keyboard.

**XSTOP ($B793).** XSTOP does the common STOP processing and then calls :LPRTOKEN($B535) to print the STOPed message. It then calls one of the error printing routines, :ERRM2 ($B974), to output the AT LINE nnn portion. The :ERRM2 routine will not print anything if this was a direct statement. When :ERRM2 is finished, it jumps back to the start of the editor.
Chapter Eight

XEND ($B78D). XEND calls the STOP routine to save the current line number. It then transfers to the start of the editor via the SNX1 entry point. This turns off the sound, closes any open IOCBs, and prints the READY message. XEND also leaves values on the 6502 CPU stack. These values are thrown away when the editor resets the stack.

END OF PROGRAM. A user may have neglected to include an END statement in his program. In this case, when Execution Control comes to the end of the Statement Table it calls XEND, and the program is terminated exactly as if the last statement in the program were an END.

Program Initiation Routines

The statements that cause a user’s program to begin execution are RUN and CONT. These statements are simulated by XRUN and XCONT.

XCONT ($B7BE). The CONT statement has no meaning when encountered as a program statement, so its execution has no effect.

When the user enters CONT as a direct statement, XCONT uses the line number that was saved in STOPLN to set Execution Control’s line parameters (STMCUR, NXTSTD, and LLNGTH). This results in the current line being the line following the one whose line number is in STOPLN. This means that any statement following STOP or END on a line will not be executed; therefore, STOP and END should always be the last statement in the line.

If we are at the end of the Statement Table, XCONT terminates as if an END statement had been encountered in the program. If there are more lines to process, XCONT returns to Execution Control, which resumes processing at the line whose address was just put into STMCUR.

XRUN ($B74D). The RUN statement comes in two formats, RUN and RUN < filespec >. In the case of RUN < filespec >, XRUN executes XLOAD to load a saved program, which replaces the current one in memory. The process then proceeds like RUN.

XRUN sets up Execution Control’s line pointers to indicate the first line in the Statement Table. It clears some flags used to control various other BASIC statements; for example, it resets STOPLN to 0. It closes all IOCBs and executes XCLR to reset all
the variables to zero and get rid of any entries in the String/Array Table or the Runtime Stack.

If there is no program, so the only thing in the Statement Table is the direct statement, then XRUN does some clean-up, prints READY, and returns to the start of the editor, which resets the 6502 CPU stack.

If there is a program, XRUN returns to Execution Control, which starts processing the first statement in the table as the current statement.

When RUN <filespec> is used as a program statement, it performs the useful function of chaining to a new program, but if RUN alone is used as a program statement, an infinite loop will probably result.

Error Handling Routine

There are other conditions besides the execution boundary statements that terminate a program's execution. The most familiar are errors.

There are two kinds of errors that can occur during execution: Input/Output errors and BASIC language errors.

Any BASIC routine that does I/O calls the IOTEST routine ($BCB3) to check the outcome of the operation. If an error that needs to be reported to the user is indicated, IOTEST gets the error number that was returned by the Operating System and joins the Error Handling Routine, ERROR ($B940), which finishes processing the error.

When a BASIC language error occurs, the error number is generated by the Error Handling Routine. This routine calculates the error by having an entry point for every BASIC language error. At each entry point, there is a 6502 instruction that increments the error number. By the time the main routine, ERROR, is reached, the error number has been generated.

The Error Handling Routine calls STOP ($B7A7) to save the line number of the line causing the error in STOPLN. It tests TRAPLN to see if errors are being TRAPed. The TRAP option is on if TRAPLN contains a valid line number. In this case, the Error Handler does some clean-up and joins XGOTO, which transfers processing to the desired line.

If the high-order byte of the line number is $80 (not a valid line number), then we are not TRAPing errors. In this case, the Error Handler prints the four-part error message, which
consists of ERROR, the error number, AT LINE, and finally the line number. If the line in error was a direct statement, the AT LINE part is not printed. The error handler resets ERRNUM to zero and is finished.

The Error Handling Routine does not do an orderly return, but jumps back to the start of the editor at the SYNTAX entry point where the 6502 stack is reset, clearing it of the now-unwanted return addresses.
Chapter Nine

Program Flow Control Statements

Execution Control always processes the statement in the Statement Table that follows the one it thinks it has just finished. This means that statements in a BASIC program are usually processed in sequential order.

Several statements, however, can change that order: GOTO, IF, TRAP, FOR, NEXT, GOSUB, RETURN, POP, and ON. They trick Execution Control by changing the parameters that it maintains.

Simple Flow Control Statements

XGOTO ($B6A3)
The simplest form of flow control transfer is the GOTO statement, simulated by the XGOTO routine.

Following the GOTO token in the tokenized line is an expression representing the line number of the statement that the user wishes to execute next. The first thing the XGOTO routine does is ask Execute Expression to evaluate the expression and convert it to a positive integer. XGOTO then calls the GETSTMT routine to find this line number in the Statement Table and change Execution Control’s line parameters to indicate this line.

If the line number does not exist, XGOTO restores the line parameters to indicate the line containing the original GOTO, and transfers to the Error Handling Routine via the ERNOLN entry point. The Error Handling Routine processes the error and jumps to the start of the editor.

If the line number was found, XGOTO jumps to the beginning of Execution Control (EXECNL) rather than returning to the point in the routine from which it was called. This leaves garbage on the 6502 CPU stack, so XGOTO first pulls the return address off the stack.
Chapter Nine

XIF ($B778)
The IF statement changes the statement flow based on a condition. The simulation routine, XIF, begins by calling a subroutine of Execute Expression to evaluate the condition. Since this is a logical (rather than an arithmetic) operation, we are only interested in whether the value is zero or non-zero. If the expression was false (non-zero), XIF modifies Execution Control’s line parameters to indicate the end of this line and then returns. Execution Control moves to the next line, skipping any remaining statements on the original IF statement line.

If the expression is true (zero), things get a little more complicated. Back during syntaxing, when a statement of the form IF <expression> THEN <statement> was encountered, the pre-compiler generated an end-of-statement token after THEN. XIF now tests for this token. If we are at the end of the statement, XIF returns to Execution Control, which processes what it thinks is the next statement in the current line, but which is actually the THEN <statement> part of the IF statement.

If XIF does not find the end-of-statement token, then the statement must have had the form IF <expression> THEN <line number>. XIF jumps to XGOTO, which finishes processing by changing Execution Control’s line parameters to indicate the new line.

XTRAP ($B7E1)
The TRAP statement does not actually change the program flow when it is executed. Instead, the XTRAP simulation routine calls a subroutine of Execute Expression to evaluate the line number and then saves the result in TRAPLN ($BC).

The program flow is changed only if there is an error. The Error Handling Routine checks TRAPLN. If it contains a valid line number, the error routine does some initial set-up and joins the XGOTO routine to transfer to the new line.

Runtime Stack Routines
The rest of the Program Flow Control Statements use the Runtime Stack. They put items on the stack, inspect them, and/or remove them from the stack.

Every item on the Runtime Stack contains a four-byte header. This header consists of a one-byte type indication, a
two-byte line number, and a one-byte displacement to the
Statement Name Token. (See pages 18-19.) The type byte is the
last byte placed on the stack for each entry. This means that the
pointer to the top of the Runtime Stack (RUNSTK) points to the
type byte of the most recent entry on the stack. A zero type
byte indicates a GOSUB-type entry. Any non-zero type byte
represents a FOR-type entry.

A GOSUB entry consists solely of the four-byte header. A
FOR entry contains twelve additional bytes: a six-byte limit
value and a six-byte step value.

Several routines are used by more than one of the
statement simulation routines.

**PSHRSTK ($B683)** This routine expands the Runtime Stack
by calling EXPLOW and then storing the type byte, line
number, and displacement of the Statement Name Token on
the stack.

**POPRSTK ($B841)** This routine makes sure there really is
an entry on the Runtime Stack. POPRSTK saves the
displacement to the statement name token in SVDISP, saves
the line number in TSLNUM, and puts the type/variable
number in the 6502 accumulator. It then removes the entry by
calling the CONTLOW routine.

**:GETTOK ($B737)** This routine first sets up Execution
Control's line parameters to point to the line whose number is
in the entry just pulled from the Runtime Stack. If the line was
found, :GETTOK updates the line parameters to indicate that
the statement causing this entry is now the current statement.
Finally, it loads the 6502 accumulator with the statement name
token from the statement that created this entry and returns to
its caller.

If the line number does not exist, :GETTOK restores the
current statement address and exits via the ERGFDEL entry
point in the Error Handling Routine.

Now let's look at the simulation routines for the statements
that utilize the Runtime Stack.

**XFOR ($B64B)**
XFOR is the name of the simulation routine which executes a
FOR statement.

In the statement FOR I = 1 TO 10 STEP 2:

*I* is the *loop control variable*
1 is its initial value
10 is the limit value
2 is the step value

XFOR calls Execute Expression, which evaluates the initial value and puts it in the loop control variable’s entry in the Variable Value Table.

Then it calls a routine to remove any currently unwanted stack entries — for example, a previous FOR statement that used the same loop control variable as this one.

XFOR calls a subroutine of Execute Expression to evaluate the limit and step values. If no step value was given, a value of 1 is assigned. It expands the Runtime Stack using EXPLOW and puts the values on the stack.

XFOR uses PSHRSTK to put the header entry on the stack. It uses the variable number of the loop control variable (machine-language ORed with $80) as the type byte. XFOR now returns to Execution Control, which processes the statement following the FOR statement.

The FOR statement does not change program flow. It just sets up an entry on the Runtime Stack so that the NEXT statement can change the flow.

XNEXT ($B6CF)
The XNEXT routine decides whether to alter the program flow, depending on the top Runtime Stack entry. XNEXT calls the POPRSTK routine repeatedly to remove four-byte header entries from the top of the stack until an entry is found whose variable number (type) matches the NEXT statement’s variable token. If the top-of-stack or GOSUB-type entry is encountered, XNEXT transfers control to an Error Handling Routine via the ERNOFOR entry point.

To compute the new value of the loop variable, XNEXT calls a subroutine of Execute Expression to retrieve the loop control variable’s current value from the Variable Value Table, then gets the step value from the Runtime Stack, and finally adds the step value to the variable value. XNEXT again calls an Execute Expression subroutine to update the variable’s value in the Variable Value Table.

XNEXT gets the limit value from the stack to determine if the variable’s value is at or past the limit. If so, XNEXT returns to Execution Control without changing the program flow, and the next sequential statement is processed.
If the variable's value has not reached the limit, XNEXT returns the entry to the Runtime Stack and changes the program flow. POPRSTK already saved the line number of the FOR statement in TSLNUM and the displacement to the statement name token in SVDISP. XNEXT calls the :GETTOK routine to indicate the FOR statement as the current statement.

If the token at the saved displacement is not a FOR statement name token, then the Error Handling Routine is given control at the ERGFDEL entry point. Otherwise, XNEXT returns to Execution Control, which starts processing with the statement following the FOR statement.

**XGOSUB ($B6A0)**

The GOSUB statement causes an entry to be made on the Runtime Stack and also changes program flow.

The XGOSUB routine puts the GOSUB-type indicator (zero) into the 6502 accumulator and calls PSHRSTK to put a four-byte header entry on the Runtime Stack for later use by the simulation routine for RETURN. XGOSUB then processes exactly like XGOTO.

**XRTN ($B719)**

The RETURN statement causes an entry to be removed from the Runtime Stack. The XRTN routine uses the information in this entry to determine what statement should be processed next.

The XRTN first calls POPRSTK to remove a GOSUB-type entry from the Runtime Stack. If there are no GOSUB entries on the stack, then the Error Handling Routine is called at ERBRTN. Otherwise, XRTN calls :GETTOK to indicate that the statement which created the Runtime Stack entry is now the current statement.

If the statement name token at the saved displacement is not the correct type, then XRTN exits via the Error Handling Routine's ERGFDEL entry point. Otherwise, control is returned to the caller. When Execution Control was the caller, then GOSUB must have created the stack entry, and processing will start at the statement following the GOSUB.

Several other statements put a GOSUB-type entry on the stack when they need to mark their place in the program. They do not affect program flow and will be discussed in later chapters.
XPOP ($B841)
The XPOP routine uses POPRSTK to remove an entry from the Runtime Stack. A user might want to do this if he decided not to RETURN from a GOSUB.

XON ($B7ED)
The ON statement comes in two versions: ON-GOTO and ON-GOSUB. Only ON-GOSUB uses the Runtime Stack.

The XON routine evaluates the variable and converts it to an integer (MOD 256). If the value is zero, XON returns to Execution Control without changing the program flow.

If the value is non-zero and this is an ON-GOSUB statement, XON puts a GOSUB-type entry on the Runtime Stack for RETURN to use later.

From this point, ON-GOSUB and ON-GOTO perform in exactly the same manner. XON uses the integer value calculated earlier to index into the tokenized statement line to the correct GOTO or GOSUB line number. If there is no line number corresponding to the index, XON returns to Execution Control without changing program flow. Otherwise, XON joins XGOTO to finish processing.
Tokenized Program
Save and Load

The tokenized program can be saved to and reloaded from a peripheral device, such as a disk or a cassette. The primary statement for saving the tokenized program is SAVE. The saved program is reloaded into RAM with the LOAD statement. The CSAVE and the CLOAD statements are special versions of SAVE and LOAD for use with a cassette.

Saved File Format
The tokenized program is completely contained within the Variable Name Table, the Variable Value Table, and the Statement Table. However, since these tables vary in size, we must also save some information about the size of the tables.

The SAVE file format is shown in Figure 10-1. The first part consists of seven fields, each of them two bytes long, which tell where each table starts or ends. Part two contains the saved program’s Variable Name Table (VNT), Variable Value Table (VVT), and Statement Table (ST).

The displacement value in all the part-one fields is actually the displacement plus 256. We must subtract 256 from each displacement value to obtain the true displacement.

The VNT starts at relative byte zero in the file’s second part. The second field in part one holds that value plus 256.

The DVVT field in part one contains the displacement, minus 256, of the VVT from the start of part two.

The DST value, minus 256, gives the displacement of the Statement Table from the start of part two.

The DEND value, minus 256, gives the end-of-file displacement from the start of part two.
The code that implements the SAVE statement starts at the XSAVE ($BB5D) label. Its first task is to open the specified output file, which it does by calling ELADVC.

The next operation is to move the first seven RAM table pointers from $80 to a temporary area at $500. While these pointers are being moved, the value contained in the first pointer is subtracted from the value in each of the seven pointers, including the first.

Since the first pointer held the absolute address of the first RAM table, this results in a list of displacements from the first RAM table to each of the other tables. These seven two-byte displacements are then written from the temporary area to the file via IO3. These are the first fourteen bytes of the SAVE file. (See Figure 10-1.)

The first RAM table is the 256-byte buffer, which will not be SAVEd. This is why the seven two-byte fields at the beginning of the SAVEd file hold values exactly 256 more than the true...
displacement of the tables they point to. (The LOAD procedure will resolve the 256-byte discrepancy.)

The next operation is to write the three needed RAM tables. The total length of these tables is determined from the value in the seventh entry in the displacement list, minus 256. To write the three entries, we point to the start of the Variable Name Table and call IO4, with the length of the three tables. This saves the second part of the file format.

The file is then closed and XSAVE returns to Execution Control.

XLOAD ($BAFB)
The LOAD statement is implemented at the XLOAD label located at $BAFB.

XLOAD first opens the specified load file for input by calling ELADVC. BASIC reads the first fourteen bytes from the file into a temporary area starting at $500. These fourteen bytes are the seven RAM table displacements created by SAVE.

The first two bytes will always be zero, according to the SAVE file format. (See Figure 10-1.) BASIC tests these two bytes for zero values. If these bytes are not zero, BASIC assumes the file is not a valid SAVE file and exits via the ERRNSF, which generates error code 21 (Load File Error).

If this is a valid SAVE file, the value in the pointer at $80 (Low Memory Address) is added to each of the seven displacements in the temporary area. These values will be the memory addresses of the three RAM tables, if and when they are read into memory.

The seventh pointer in the temporary area contains the address where the end of the Statement Table will be. If this address exceeds the current system high memory value, the routine exits via ERRPTL, which generates error code 19 (Load Program Too Big).

If the program will fit, the seven addresses are moved from the temporary area to the RAM table pointers at $80. The second part of the file is then loaded into the area now pointed to by the Variable Name Table pointer $82. The file is closed, CLR is executed, and a test for RUN is made.

If RUN called XLOAD, then a value of $FF was pushed onto the CPU stack. If RUN did not call XLOAD, then $00 was pushed onto the CPU stack. If RUN was the caller, then an RTS is done.
If XLOAD was entered as a result of a LOAD or CLOAD statement, then XLOAD exits directly to the Program Editor, not to Execution Control.

**CSAVE and CLOAD**
The CSAVE and CLOAD statements are special forms of SAVE and LOAD. These two statements assume that the SAVE/LOAD device is the cassette device.

CSAVE is not quite the same as SAVE "C:". Using SAVE with the "C:" device name will cause the program to be saved using long cassette inter-record gaps. This is a time waster, and CSAVE uses short inter-record gaps.

CSAVE starts at XCSAVE ($BBAC). CLOAD starts at XCLOAD ($BBA4).
LIST can be used to store a program on an external device and ENTER can retrieve it. The difference between LOAD-SAVE and LIST-ENTER is that LOAD-SAVE deals with the tokenized program, while LIST-ENTER deals with the program in its source (ATASCII) form.

The ENTER Statement
BASIC is in ENTER mode whenever a program is not RUNning. By default the Program Editor looks for lines to be ENTERed from the keyboard, but the editor handles all ENTERed lines alike, whether they come from the keyboard or not.

The Enter Device
To accomplish transparency of all input data (not just ENTERed lines), BASIC maintains an enter device indicator, ENTDTD ($B4). When a BASIC routine (for example, the INPUT simulation routine) needs data, an I/O operation is done to the IOCB specified in ENTDTD. When the value in ENTDTD is zero, indicating IOCB 0, input will come from the keyboard. When data is to come from some other device, ENTDTD contains a number indicating the corresponding IOCB. During coldstart initialization, the enter device is set to IOCB 0. It is also reset to 0 at various other times.

XENTER ($BACB)
The XENTER routine is called by Execution Control to simulate the ENTER statement. XENTER opens IOCB 7 for input using the specified <filespec>, stores a 7 in the enter device ENTDTD, and then jumps to the start of the editor.

Entering from a Device
When the Program Editor asks GLGO, the get line routine ($BA92), for the next line, GLGO tells CIO to get a line from the
device specified in ENTDTD — in this case, from IOC B 7. The editor continues to process lines from IOC B 7 until an end-of-file error occurs. The IOTEST routine detects the EOF condition, sees that we are using IOC B 7 for ENTER, closes device 7, and jumps to SNX2 to reset the enter device (ENTDTD) to 0 and print the READY message before restarting at the beginning of the editor.

The LIST Statement
The routine which simulates the LIST statement, XLIST, is actually another example of a language translator, complete with symbols and symbol-combining rules. XLIST translates the tokens generated by Atari BASIC back into the semi-English BASIC statements in ATASCII. This translation is a much simpler task than the one done by the pre-compiler, since XLIST can assume that the statement to be translated is syntactically correct. All that is required is to translate the tokens and insert blanks in the appropriate places.

The List Device
BASIC maintains a list device indicator, LISTDTD ($B5), similar to the enter device indicator discussed earlier. When a BASIC routine wants to output some data (an error message, for example), the I/O operation is done to the device (IOCB) specified in LISTDTD.

During coldstart initialization and at various other times, LISTDTD is set to zero, representing IOC B 0, the editor, which will place the output on the screen. Routines such as XPRINT or XLIST can change the LIST device to indicate some other IOC B. Thus the majority of the BASIC routines need not be concerned about the output’s destination.

Remember that IOC B 0 is always open to the editor, which gets input from the keyboard and outputs to the screen. IOC B 6 is the S: device, the direct access to graphics screen, which is used in GRAPHICS statements. Atari BASIC uses IOC B 7 for I/O commands that allow different devices, like SAVE, LOAD, ENTER, and LIST.

XLIST ($B483)
The XLIST routine considers the output’s destination in its initialization process and then forgets about it. It looks at the first expression in the tokenized line. If it is the <filespec>
string, XLIST calls a routine to open the specified device using IOCB 7 and to store a 7 in LISTDTD. All of XLIST's other processing is exactly the same, regardless of the LISTed data's final destination.

XLIST marks its place in the Statement Table by calling a subroutine of XGOSUB to put a GOSUB type entry on the Runtime Stack. Then XLIST steps through the Statement Table in the same way that Execution Control does, using Execution Control's line parameters and subroutines. When XLIST is finished, Execution Control takes the entry off the Runtime Stack and continues.

The XLIST routine, assuming it is to LIST all program statements, sets default starting and ending line numbers of 0 (in TSLNUM) and $7FFF (in LELNUM).

XLIST then determines whether line numbers were specified in the tokenized line that contained the LIST statement. XLIST compares the current index into the line (STINDEX) to the displacement to the next statement (NXTSTD). If STINDEX is not pointing to the next statement, at least one line number is specified. In this case, XLIST calls a subroutine of Execute Expression to evaluate the line number and convert it to a positive integer, which XLIST stores in TSLNUM as the starting line number.

If a second line number is specified, XLIST calls Execute Expression again and stores the value in LELNUM as the final line to LIST. If there is no second line number, then XLIST makes the ending line number equal to the starting line number, and only one line will be LISTed. If no line numbers were present, then TSLNUM and LELNUM still contain their default values, and all the program lines will be LISTed.

XLIST gets the first line to be LISTed by calling the Execution Control subroutine GETSTMT to initialize the line parameters to correspond to the line number in TSLNUM. If we are not at the end of the Statement Table, and if the current line's number is less than or equal to the final line number to be LISTed, XLIST calls a subroutine :LLINE to list the line.

After LISTing the line, XLIST calls Execution Control's subroutines to point to the next line. LISTing continues in this manner until the end of the Statement Table is reached or until the final line specified has been printed.

When XLIST is finished, it exits via XRTN at $B719, which makes the LIST statement the current statement again and then returns to Execution Control.
LIST Subroutines

:LLINE ($B55C)
The :LLINE routine LISTs the current line (the line whose address is in STMCUR).

:LLINE gets the line number from the beginning of the tokenized line. The floating point package is called to convert the integer to floating point and then to printable ATASCII. The result is stored in the buffer indicated by INBUFF. :LLINE calls a subroutine to print the line number and then a blank.

For every statement in the line, :LLINE sets STINDEX to point to the statement name token and calls the :LSTMT routine ($B590) to LIST the statement. When all statements have been LISTed, :LLINE returns to its caller, XLIST.

:LSTMT ($B590)
The :LSTMT routine LISTs the statement which starts at the current displacement (in STINDEX) into the current line. This routine does the actual language translation from tokens to BASIC statements.

:LSTMT uses two subroutines, :LGCT and :LGNT, to get the current and next token, respectively. If the end of the statement has been reached, these routines both pull the return address of their caller off the 6502 CPU stack and return to :LSTMT's caller, :LLINE. Otherwise, they return the requested token from the tokenized statement line.

The first token in a statement is the statement name token. :LSTMT calls a routine which prints the corresponding statement name by calling :LSCAN to find the entry and :LPRTOKEN to print it.

In the discussion of the Program Editor we saw that an erroneous statement was given a statement name of ERROR and saved in the Statement Table. If the current statement is this ERROR statement or is REM or DATA, :LSTMT picks up each remaining character in the statement and calls PRCHAR ($BA9F) to print the character.

Each type of token is handled differently. :LSTMT determines the type (variable, numeric constant, string constant, or operator) and goes to the proper code to translate it.

Variable Token. A variable token has a value greater than or equal to $80. When :LSTMT encounters a variable token, it
turns off the most significant bit to get an index into the Variable Name Table. :LSTMT asks the :LSCAN routine to get the address of this entry. :LSTMT then calls :LPRTOKEN ($B535) to print the variable name. If the last character of the name is (, the next token is an array left parenthesis operator, and :LSTMT skips it.

**Numeric Constant Token.** A numeric constant is indicated by a token of $0E. The next six bytes are a floating point number. :LSTMT moves the numeric constant from the tokenized line to FRO ($D4) and asks the floating point package to convert it to ATASCII. The result is in a buffer pointed to by INBUFF. :LSTMT moves the address of the ATASCII number to SRCADR and tells :LPRTOKEN to print it.

**String Constant Token.** A string constant is indicated by a token of $0F. The next byte is the length of the string followed by the actual string data. Since the double quotes are not stored with a string constant, :LSTMT calls PRCHAR ($BA9F) to print the leading double quote. The string length tells :LSTMT how many following characters to print without translation. :LSTMT repeatedly gets a character and calls PRCHAR to print it until the whole string constant has been processed. It then asks PRCHAR to print the ending double quote.

**Operator Token.** An operator token is any token greater than or equal to $10 and less than $80. By subtracting $10 from the token value, :LSTMT creates an index into the Operator Name Table. :LSTMT calls :LSCAN to find the address of this entry. If the operator is a function (token value greater than or equal to $3D), :LPROTOKEN is called to print it. If this operator is not a function but its name is alphabetic (such as AND), the name is printed with a preceding and following blank. Otherwise, :LPRTOKEN is called to print just the operator name.

**:LSCAN ($B50C)**
This routine scans a table until it finds the translation of a token into an ATASCII name. A token’s value is based on its table entry number; therefore, the entry number can be derived by modifying the token. For example, a variable token is created by machine-language ORing the table entry number of the variable name with $80. The entry number can be produced by ANDing out the high-order bit of the token. It is this entry number, stored in SCANT, that the :LSCAN routine uses.
The tables scanned by :LSCAN have a definite structure. Each entry consists of a fixed length portion followed by a variable length ATASCII portion. The last character in the ATASCII portion has the high-order bit on. Using these facts, :LSCAN finds the entry corresponding to the entry number in SCANT and puts the address of the ATASCII portion in SCRADR.

:LPRTOKEN ($B535)
This routine's task is to print the string of ATASCII characters whose address is in SCRADR. :LPRTOKEN makes sure the most significant bit is off (except for a carriage return) and prints the characters one at a time until it has printed the last character in the string (the one with its most significant bit on).
Atari Hardware Control Statements

The Atari Hardware Control Statements allow easy access to some of the computer’s graphics and audio capabilities. The statements in this group are COLOR, GRAPHICS, PLOT, POSITION, DRAWT0, SETCOLOR, LOCATE, and SOUND.

**XGR ($BA50)**

The GRAPHICS statement determines the current graphics mode. The XGR simulation routine executes the GRAPHICS statement. The XGR routine first closes IOCB 6. It then calls an Execute Expression subroutine to evaluate the graphics mode value and convert it to an integer.

XGR sets up to open the screen by putting the address of a string "S:" into INBUFF. It creates an AUX1 and AUX2 byte from the graphics mode integer. XGR calls a BASIC I/O routine which sets up IOCB 6 and calls CIO to open the screen for the specified graphics mode. Like all BASIC routines that do I/O, XGR jumps to the IOTEST routine, which determines what to do next based on the outcome of the I/O.

**XCOLOR ($BA29)**

The COLOR statement is simulated by the XCOLOR routine. XCOLOR calls a subroutine of Execute Expression to evaluate the color value and convert it to an integer. XCOLOR saves this value (MOD 256) in BASIC memory location COLOR ($C8). This value is later retrieved by XPLOT and XDRAWT0.

**XSETCOLOR ($B9B7)**

The routine that simulates the SETCOLOR statement, XSETCOLOR, calls a subroutine of Execute Expression to evaluate the color register specified in the tokenized line. The Execute Expression routine produces a one-byte integer. If the value is not less than 5 (the number of color registers), XSETCOLOR exits via the Error Handling Routine at entry point ERVAL. Otherwise, it calls Execute Expression to get two more integers from the tokenized line.
To calculate the color value, XSETCOLOR multiplies the first integer (MOD 256) by 16 and adds the second (MOD 256). Since the operating system's five color registers are in consecutive locations starting at $2C4, XSETCOLOR uses the register value specified as an index to the proper register location and stores the color value there.

**XPOS ($BA16)**
The POSITION statement, which specifies the X and Y coordinates of the graphics cursor, is simulated by the XPOS routine.

XPOS uses a subroutine of Execute Expression to evaluate the X coordinate of the graphics window cursor and convert it to an integer value. The two-byte result is stored in the operating system's X screen coordinate location (SCRX at $55). This is the column number or horizontal position of the cursor.

XPOS then calls another Execute Expression subroutine to evaluate the Y coordinate and convert it to a one-byte integer. The result is stored in the Y screen coordinate location (SCRY at $54). This is the row number, or vertical position.

**XLOCATE ($BC95)**
XLOCATE, which simulates the LOCATE statement, first calls XPOS to set up the X and Y screen coordinates. Next it initializes IOCBO and joins a subroutine of XGET to do the actual I/O required to get the screen data into the variable specified.

**XPLOT ($BA76)**
XPLOT, which simulates the PLOT statement, first calls XPOS to set the X and Y coordinates of the graphics cursor. XPLOT gets the value that was saved in COLOR ($C8) and joins a PUT subroutine (PRCX at $BAA1) to do the I/O to IOCBO (the screen).

**XDRAWTO ($BA31)**
The XDRAWTO routine draws a line from the current X,Y screen coordinates to the X,Y coordinates specified in the statement. The routine calls XPOS to set the new X,Y coordinates. It places the value from BASIC's memory location COLOR into OS location SVCOLOR ($2FB). XDRAWTO does some initialization of IOCBO specifying the draw command ($11). It then calls a BASIC I/O routine which finishes the
initialization of IOCB 6 and calls CIO to draw the line. Finally, XDRAWTO jumps to the IOTEST routine, which will determine what to do next based on the outcome of the I/O.

**X SOUND ($B9DD)**
The Atari computer hardware uses a set of memory locations to control sound capabilities. The SOUND statement gives the user access to some of these capabilities. The X SOUND routine, which simulates the SOUND statement, places fixed values in some of the sound locations and user specified values in others.

The X SOUND routine uses Execute Expression to get four integer values from the tokenized statement line. If the first integer (voice) is greater than or equal to 4, the Error Handling Routine is invoked at ERVAL.

The OS audio control bits are all turned off by storing a 0 into $D208. Any bits left on from previous serial port usage are cleared by storing 3 in $D20F.

The Atari has four sound registers (one for each voice) starting at $D200. The first byte of each two-byte register determines the pitch (frequency). In the second byte, the four most significant bits are the distortion, and the four least significant bits are the volume.

The voice value mentioned earlier is multiplied by 2 and used as an index into the sound registers. The second value from the tokenized line is stored as the pitch in the first byte of one of the registers ($D200, $D202, $D204, or $D206), depending on the voice index. The third value from the tokenized line is multiplied by 16 and the fourth value is added to it to create the value to be stored as distortion/volume. The voice, times 2, is again used as an index to store this value in the second byte of a sound register ($D201, $D203, $D205, or $D207). The X SOUND routine then returns to Execution Control.
Chapter Thirteen

External Data

I/O Statements

The external data I/O statements allow data which is not part of
the BASIC source program to flow into and out of BASIC.
External data can come from the keyboard, a disk, or a cassette.
BASIC can also create external information by sending data to
external devices such as the screen, a printer, or a disk.

The INPUT and GET statements are the primary
statements used for obtaining information from external
devices. The PRINT and PUT statements are the primary
statements for sending data to external devices.

XIO, LPRINT, OPEN, CLOSE, NOTE, POINT and
STATUS are specialized I/O statements. LPRINT is used to
print a single line to the "P:" device. The other statements
assist in the I/O process.

XINPUT ($B316)
The execution of the INPUT statement starts at XINPUT
($B316).

Getting the Input Line. The first action of XINPUT is to
read a line of data from the indicated device. A line is any
combination of up to 255 characters terminated by the EOL
character ($9B). This line will be read into the buffer located at
$580.

If the INPUT statement contained was followed by
# <expression>, the data will be read from the IOCB whose
number was specified by <expression>. If there was no
# <expression>, IOCB 0 will be used. IOCB 0 is the screen
editor and keyboard device (E:). If IOCB 0 is indicated, the
prompt character (?) will be displayed before the input line
request is made; otherwise, no prompt is displayed.

Line Processing. Once the line has been read into the
buffer, processing of the data in that line starts at XINA
($B335). The input line data is processed according to the
tokens in the INPUT (or READ) statements. These tokens are
numeric or string variables separated by commas.
Chapter Thirteen

Processing a Numeric Variable. If the new token is a numeric variable, the CVAFP routine is called to convert the next characters in the input line to a floating point number. If this conversion does not report an error, and if the next input line character is a comma or an EOL, the floating point value is processed.

The processing of a valid numeric input value consists of calling RTNVAR to return the variable and its new value to the Variable Value Table.

If there is an error, INPUT processing is aborted via the ERRINP routine. If there is no error, but the user has hit BREAK, the process is aborted via XSTOP. If there is no abort, XINX ($B389) is called to continue with INPUT’s next task.

Processing a String Variable. If the next statement token is a string variable, it is processed at XISTR ($B35E). This routine is also used by the READ statement. If the calling statement is INPUT, then all input line characters from the current character up to but not including the EOL character are considered to be part of the input string data. If the routine was called by READ, all characters up to but not including the next comma or EOL are considered to be part of the input string.

The process of assigning the data to the string variable is handled by calling RISASN ($B386). If RISASN does not abort the process because of an error like DIMENSION TOO SMALL, XINX is called to continue with INPUT’s next task.

XINX. The XINX ($B389) routine is entered after each variable token in an INPUT or a READ statement is processed.

If the next token in the statement is an EOL, the INPUT/READ statement processing terminates at XIRTS ($B3A1). XIRTS restores the line buffer pointer ($80) to the RAM table buffer. It then restores the enter device to IOCB 0 (in case it had been changed to some other input device). Finally, XIRTS executes an RTS instruction.

If the next INPUT/READ statement token is a comma, more input data is needed. If the next input line character is an EOL, another input line is obtained. If the statement was INPUT, the new line is obtained by entering XIN0 ($B326). If the statement was READ, the new line is obtained by entering XRD3 ($B2D0).

The processing of the next INPUT/READ statement variable token continues at XINA.
XGET ($BC7F)
The GET statement obtains one character from some specified
device and assigns that character to a scalar (non-array)
numeric variable.

The execution of GET starts at XGET ($BC7F) with a call to
GIODVC. GIODVC will set the I/O device to whatever number
is specified in the #<expression> or to IOCB zero if no
#<expression> was specified. (If the device is IOCB 0 (E::), the
user must type RETURN to force E: to terminate the input.)

The single character is obtained by calling I03. The
character is assigned to the numeric variable by calling ISVAR1
($BD2D). ISVAR1 also terminates the GET statement
processing.

PRINT
The PRINT statement is used to transmit text data to an
external device. The arguments in the PRINT statement are a
list of numeric and/or string expressions separated by commas
or semicolons. If the argument is numeric, the floating point
value is converted to text form. If the argument is a string, the
string value is transmitted as is.

If an argument separator is a comma, the arguments are
output in tabular fashion: each new argument starts at the next
tab stop in the output line, with blanks separating the
arguments.

If the argument separator is a semicolon, the transmitted
arguments are appended to each other without separation.

The transmitted line is terminated with an EOL, unless a
semicolon or comma directly precedes the statement's EOL or
statement separator (:).

XPRINT ($B3B6). The PRINT routine begins at XPRINT
($B3B6). The tab value is maintained in the PTABW ($C9) cell.
The cell is initialized with a value of ten during BASIC's cold
start, so that commas in the PRINT line cause each argument to
be displaced ten positions after the beginning of the last
argument. The user may POKE PTABW to set a different tab
value.

XPRINT copies PTABW to SCANT ($AF). SCANT will be
used to contain the next multiple-of-PTABW output line
displacement — the column number of the next tab stop.

COX is initialized to zero and is used to maintain the
current output column or displacement.
Chapter Thirteen

XPR0. XPRINT examines the next statement token at XPR0 ($B3BE), classifies it, and executes the proper routine.

# Token. If the next token is #, XPRIOD ($B437) is entered. This routine modifies the list device to the device specified in the # < expression >. XPR0 is then entered to process the next token.

, Token. The XPTAB ($B419) routine is called to process the , token. Its job is to tab to the next tab column.

   If COX (the current column) is greater than SCANT, we must skip to the next available tab position. This is done by continuously adding PTABW to SCANT until COX is less than or equal to SCANT. When COX is less than SCANT, blanks ($20) are transmitted to the output device until COX is equal to SCANT.

   The next token is then examined at XPR0.

EOL and : Tokens. The XPEOS ($B446) routine is entered for EOL and : tokens. If the previous token was a ; or , token, PRINT exits at XPRTN ($B458). If the previous token was not a ; or , token, an EOL character is transmitted before exiting via XPRTN.

; Token. No special action is taken for the ; token except to go to XPR0 to examine the next token.

Numbers and Strings. If the next token is not one of the above tokens, Execute Expression is called to evaluate the expression. The resultant value is popped from the argument stack and its type is tested for a number or a string.

   If the argument popped was numeric, it will be converted to text form by calling CVFASC. The resulting text is transmitted to the output device from the buffer pointed to by INBUFF ($F3). XPR0 is then entered to process the next token.

   If the argument popped was a string, it will be transmitted to the output device by the code starting at :XPSTR ($B3F8). This code examines the argument parameters to determine the current length of the string. When the string has been transmitted, XPR0 is entered to process the next token.

XLPRINT ($B464)
LPRINT, a special form of the PRINT statement, is used to print a line to the printer device (P:).
The XLPRINT routine starts at $B464 by opening IOCB 7 for output to the P: device. XPRINT is then called to do the printing. When the XPRINT is done, IOCB 7 is closed via CLSYS1 and LPRINT is terminated.

**XPUT ($BC72)**
The PUT statement sends a single byte from the expression in the PUT statement to a specified external device.

Processing starts at XPUT ($BC72) with a call to GIODVC. GIODVC sets the I/O device to the IOCB specified in # <expression>. If a #<expression> does not exist, the device will be set to IOCB zero (E:).

The routine then calls GETINT to execute PUT’s expression and convert the resulting value to a two-byte integer. The least significant byte of this integer is then sent to the PUT device via PRCX. PRCX also terminates the PUT processing.

**XXIO ($BBE5)**
The XIO statement, a general purpose I/O statement, is intended to be used when no other BASIC I/O statement will serve the requirements. The XIO parameters are an IOCB I/O command, an IOCB specifying expression, an AUX1 value, an AUX2 value, and finally a string expression to be used as a filespec parameter.

XIO starts at XXIO ($BBE5) with a call to GIOCMD. GIOCMD gets the IOCB command parameter. XIO then continues at XOP1 in the OPEN statement code.

**XOPEN ($BBEB)**
The OPEN statement is used to open an external device for input and/or output. OPEN has a # <expression>, the open type parameter (AUX1), an AUX2 parameter, and a string expression to be used as a filespec.

OPEN starts at XOPEN at $BBEB. It loads the open command code into the A register and continues at XOP1.

**XOP1.** XOP1 continues the OPEN and XIO statement processing. It starts at $BBED by storing the A register into the IOCMD cell. Next it obtains the AUX1 (open type) and AUX2 values from the statement.

The next parameter is the filespec string. In order to insure that the filespec has a proper terminator, SETSEOL is called to place a temporary EOL at the end of the string.
Chapter Thirteen

The XIO or OPEN command is then executed via a call to IO1. When IO1 returns, the temporary EOL at the end of the string is replaced with its previous value by calling RSTSEOL. OPEN and XIO terminate by calling IOTEST to insure that the command was executed without error.

**XCLOSE ($BC1B)**
The CLOSE statement, which closes the specified device, starts at XCLOSE ($BC1B). It loads the IOCB close command code into the A register and continues at GDVCIO.

**GDVCIO.** GDVCIO ($BC1D) is used for general purpose device I/O. It stores the A register into the IOCMD cell, calls GIODVC to get the device from #<expression>, then calls IO7 to execute the I/O. When IO7 returns, IOTEST is called to test the results of the I/O and terminate the routine.

**XSTATUS ($BC28)**
The STATUS statement executes the IOCB status command. Processing starts at XSTATUS ($BC28) by calling GIODVC to get the device number from #<expression>. It then calls IO8 with the status command in the A register. When IO8 returns, the status returned in the IOCB status cell is assigned to the variable specified in the STATUS statement by calling ISVAR1. ISVAR1 also terminates the STATUS statement processing.

**XNOTE ($BC36)**
The NOTE statement is used specifically for disk random access. NOTE executes the Disk Device Dependent Note Command, $26, which returns two values representing the current position within the file for which the IOCB is open.

NOTE begins at XNOTE at $BC36. The code loads the command value, $26, into the A register and calls GDVCIO to do the I/O operation. When GDVCIO returns, the values are moved from AUX3 and AUX4 to the first variable in the NOTE statement. The next variable is assigned the value from AUX5.

**XPOINT ($BC4D)**
The POINT statement is used to position a disk file to a previously NOTEd location. Processing starts at XPOINT ($BC4D). This routine converts the first POINT parameter to an integer and stores the value in AUX3 and AUX4. The second parameter is then converted to an integer and its value stored
in AUX5. The POINT command, $25, is executed by calling GDIO1, which is part of GDVCIO.

**Miscellaneous I/O Subroutines**

**IOTEST.** IOTEST ($BCB3) is a general purpose routine that examines the results of an I/O operation. If the I/O processing has returned an error, IOTEST processes that error.

IOTEST starts by calling LDIOSTA to get the status byte from the IOCB that performed the last I/O operation. If the byte value is positive (less than 128), IOTEST returns to the caller.

If the status byte is negative, the I/O operation was abnormal and processing continues at SICKIO.

If the I/O aborted due to a BREAK key depression, BRKBYT ($11) is set to zero to indicate BREAK. If a LOAD was in progress when BREAK was hit, exit is via COLDSTART; otherwise IOTEST returns to its caller.

If the error was not from IOCB 7 (the device BASIC uses), the error status value is stored in ERRNUM and ERROR is called to print the error message and abort program execution.

If the error was from IOCB 7, then IOCB 7 is closed and ERROR is called with the error status value in ERRNUM — unless ENTER was being executed, and the error was an end-of-file error. In this case, IOCB 7 is closed, the enter device is reset to IOCB 0, and SNX2 is called to return control to the Program Editor.

**I/O Call Routine.** All I/O is initiated from the routine starting at IO1 ($BD0A). This routine has eight entry points, IO1 through IO8, each of which stores predetermined values in an IOCB. All IO\textsubscript{n} entry points assume that the X register contains the IOCB value, times 16.

- IO1 sets the buffer length to 255.
- IO2 sets the buffer length to zero.
- IO3 sets the buffer length to the value in the Y register plus a most-significant length byte of zero.
- IO4 sets the buffer length from the values in the Y,A register pair, with the A register being the most-significant value.
- IO5 sets the buffer address from the value in the INBUFF cell ($F3).
- IO6 sets the buffer address from the Y,A register pair. The A register contains the most significant byte.
Chapter Thirteen

IO7 sets the I/O command value from the value in the IOCMD cell.
IO8 sets the I/O command from the value in the A register.
All of this is followed by a call to the operating system CIO entry point. This call executes the I/O. When CIO returns, the general I/O routine returns to its caller.
Chapter Fourteen

Internal I/O Statements

The READ, DATA, and RESTORE statements work together to allow the BASIC user to pass predetermined information to his or her program. This is, in a sense, internal I/O.

**XDATA ($A9E7)**
The information to be passed to the BASIC program is stored in one or more DATA statements. A DATA statement can occur any place in the program, but execution of a DATA statement has no effect.

When Execution Control encounters a DATA statement, it expects to process this statement just like any other. Therefore an XDATA routine is called, but XDATA simply returns to Execution Control.

**XREAD ($B283)**
The XREAD routine must search the Statement Table to find DATA. It uses Execution Control’s subroutines and line parameters to do this. When XREAD is done, it must restore the line parameters to point to the READ statement. In order to mark its place in the Statement Table, XREAD calls a subroutine of XGOSUB to put a GOSUB-type entry on the Runtime Stack.

The BASIC program may need to READ some DATA, do some other processing, and then READ more DATA. Therefore, XREAD needs to keep track of just where it is in which DATA statement. There are two parameters that provide for this. DATALN ($B7) contains the line number at which to start the search for the next DATA statement. DATAD ($B6) contains the displacement of the next DATA element in the DATALN line. Both values are set to zero as part of RUN and CLR statement processing.

XREAD calls Execution Control’s subroutine GETSTMT to get the line whose number is stored in DATALN. If this is the first READ in the program and a RESTORE has not set a
different line number, DATALN contains zero, and GETSTMT will get the first line in the program. On subsequent READs, GETSTMT gets the last DATA statement that was processed by the previous READ.

After getting its first line, XREAD calls the XRTN routine to restore Execution Control’s line parameters.

The current line number is stored in DATALN. XREAD steps through the line, statement by statement, looking for a DATA statement. If the line contains no DATA statement, then subsequent lines and statements are examined until a DATA statement is found.

When a DATA statement has been found, XREAD inspects the elements of the DATA statement until it finds the element whose displacement is in DATAD.

If no DATA is found, XREAD exits via the ERROOD entry point in the Error Handling Routine. Otherwise, a flag is set to indicate that a READ is being done, and XREAD joins XINPUT at :XINA. XINPUT handles the assignment of the DATA values to the variables. (See Chapter 13.)

**XREST ($B26B)**
The RESTORE statement allows the BASIC user to re-READ a DATA statement or change the order in which the DATA statements are processed. The XREST routine simulates RESTORE.

XREST sets DATALN to the line number given, or to zero if no line number is specified. It sets DATAD to zero, so that the next READ after a RESTORE will start at the first element in the DATA line specified in DATALN.
Chapter Fifteen

Miscellaneous Statements

**XDEG ($B261) and XRAD ($B266)**
The transcendental functions such as SIN or COS will work with either degrees or radians, depending on the setting of RADFLG ($FB). The DEG and RAD statements cause RADFLG to be set. These statements are simulated by the XDEG and XRAD routines, respectively.

The XDEG routine stores a six in RADFLG. XRAD sets it to zero. These particular values were chosen because they aid the transcendental functions in their calculations.

RADFLG is set to zero during BASIC's initialization process and also during simulation of the RUN statement.

**XPOKE ($B24C)**
The POKE statement is simulated by the XPOKE routine. XPOKE calls a subroutine of Execute Expression to get the address and data integers from the tokenized line. XPOKE then stores the data at the specified address.

**XBYE ($A9E8)**
The XBYE routine simulates the BYE statement. XBYE closes all IOCBs (devices and files) and then jumps to location $E471 in the Operating System. This ends BASIC and causes the memo pad to be displayed.

**XDOS ($A9EE)**
The DOS statement is simulated by the XDOS routine. The XDOS routine closes all IOCBs and jumps to whatever address is stored in location $0A. This will be the address of DOS if DOS has been loaded. If DOS has not been loaded, $0A will point to the memo pad.

**XLET ($AAE0)**
The LET and implied LET statements assign values to variables. They both invoke the XLET routine, which consists of the Execute Expression routines. (See Chapter 7.)
XREM ($A9E7)
The REM statement is for documentation purposes only and has no effect on the running program. The routine which simulates REM, XREM, simply executes an RTS instruction to return to Execution Control.

XERR ($B91E)
When a line containing a syntax error is entered, it is given a special statement name token to indicate the error. The entire line is flagged as erroneous no matter how many previously good statements are in the line. The line is then stored in the Statement Table.

The error statement is processed just like any other. Execution Control calls a routine, XERR, which is one of the entry points to the Error Handling Routine. It causes error 17 (EXECUTION OF GARBAGE).

XDIM ($B1D9)
The DIMension statement, simulated by the XDIM routine, reserves space in the String/Array Table for the DIMensioned variable.

The XDIM routine calls Execute Expression to get the variable to be DIMensioned from the Variable Value Table. The variable entry is put into a work area. In the process, Execute Expression gets the first and second DIMension values and sets a default of zero if only one value is specified.

XDIM checks to see if the variable has already been DIMensioned. If the variable was already DIMensioned, XDIM exits via the ERRDIM entry point in the Error Handling Routine. If not, a bit is set in the variable type byte in the work area entry to mark this variable as DIMensioned.

Next, XDIM calculates the amount of space required. This calculation is handled differently for strings and arrays.

DIMensioning an Array. XDIM first increments both dimension values by one and then multiplies them together to get the number of elements in the array. XDIM multiplies the result by 6 (the length of a floating point number) to get the number of bytes required. EXPAND is called to expand the String/Array Table by that amount.

XDIM must finish building the variable entry in the work area. It stores the first and second dimension values in the entry. It also stores the array’s displacement into the
String/Array Table. It then calls an Execute Expression subroutine to return the variable to the Variable Value Table. (See Chapter 3.)

**DIMensioning a String.** Reserving space for a string in the String/Array Table is much simpler. XDIM merely calls the EXPAND routine to expand by the user-specified size.

XDIM must also build the Variable Value Table entry in the work area. It sets the current length to 0 and the maximum length to the DIMensioned value. The displacement of the string into the String/Array Table is also stored in the variable. XDIM then calls a subroutine of Execute Expression to return the variable entry to the Variable Value Table. (See Chapter 3.)
Chapter Sixteen

Initialization

When the Atari computer is powered up with the BASIC cartridge in place, the operating system does some processing and then jumps to a BASIC routine. Between the time that BASIC first gets control and the time it prints the READY message, initialization takes place. This initialization is called a cold start. No data or tables are preserved during a cold start.

Initialization is repeated if things go terribly awry. For example, if there is an I/O error while executing a LOAD statement, BASIC is totally confused. It gives up and begins all over again with the COLDSTART routine.

Sometimes a less drastic partial initialization is necessary. This process is handled by the WARMSTART routine, in which some tables are preserved.

Entering the NEW statement, simulated by the XNEW routine, has almost the same effect as a cold start.

COLDSTART ($A000)
Two flags, LOADFLG and WARMFLG, are used to determine if a cold or warm start is required.

The load flag, LOADFLG ($CA), is zero except during the execution of a LOAD statement. The XLOAD routine sets the flag to non-zero when it starts processing and resets it to zero when it finishes. If an I/O error occurs during that interval, IOTEST notes that LOADFLG is non-zero and jumps to COLDSTART.

The warm-start flag, WARMFLG ($08), is never set by BASIC. It is set by some other routine, such as the operating system or DOS. If WARMFLG is zero, a cold start is done. If it is non-zero, a warm start is done. During its power-up processing, before BASIC is given control, OS sets WARMFLG to zero to request a cold start. During System Reset processing, OS sets the flag to non-zero, indicating a warm start is desired.

If DOS has loaded any data into BASIC's program area during its processing, it will request a cold start.

The COLDSTART routine checks both WARMFLG and LOADFLG to determine whether to do a cold or warm start. If a cold start is required, COLDSTART initializes the 6502 CPU.
Chapter Sixteen

stack and clears the decimal flag. The rest of its processing is exactly the same as if the NEW statement had been entered.

**XNEW ($A00C)**
The NEW statement is simulated by the XNEW routine. XNEW resets the load flag, LOADFLG, to zero. It initializes the zero-page pointers to BASIC’s RAM tables. It reserves 256 bytes at the low memory address for the multipurpose buffer and stores its address in the zero-page pointer located at $80. Since none of the RAM tables are to retain any data, their zero-page pointers ($82 through $90) are all set to low memory plus 256.

The Variable Name Table is expanded by one byte, which is set to zero. This creates a dummy end-of-table entry.

The Statement Table is expanded by three bytes. The line number of the direct statement ($8000) is stored there along with the length (three). This marks the end of the Statement Table.

A default tab value of 10 is set for the PRINT statement.

**WARMSTART ($A04D)**
A warm start is the least drastic of the three types of initialization. Everything the WARMSTART routine does is also done by COLDSTART and XNEW.

The stop line number (STOPLN), the error number (ERRNUM), and the DATA parameters (DATALN and DATAD) are all set to zero. The RADFLG flag is set to zero, indicating that transcendental functions are working in radians. The break byte (BRKBYT) is set off and $FF is stored in TRAPLN to indicate that errors are not being trapped.

All IOCBs (devices and files) are closed.

The enter and list devices (ENTDTD and LISTDTD) are set to zero to indicate the keyboard and the screen, respectively.

Finally, the READY message is printed and control passes to the Program Editor.
Introduction to Part Two

Congratulations! If you have read all of Part 1, you are through the hard stuff. In Part 2, we hope to teach you how to use at least some of the abundance of information presented in the Source Listing and in Part 1. In particular, we will show you how to examine the various RAM and ROM tables used by BASIC.

The examples and suggestions will be written in Atari BASIC. But those of you who are true-blue assembly language fanatics should have little trouble translating the concepts to machine code, especially with the source listing to guide you.

Would that we could present an example program or concept for each possible aspect of the BASIC interpreter, but space does not allow it — nor would it be appropriate. For example, although we will present here a program to list all keywords and token values used by BASIC, we will *not* explore the results (usually disastrous) of changing token values within a BASIC program.

Part 2 begins with a pair of introductory chapters. If you are experienced at hexadecimal-to-decimal conversions and with the concepts of word and byte PEEKs and POKEs, you may wish to skip directly to Chapter 3.
Hexadecimal Numbers

The word hexadecimal means, literally, "of six and ten." It implies, however, a number notation which uses 16 as its base instead of 10. Hexadecimal notation is used as a sort of shorthand for the eight-digit binary numbers that the 6502 understands. If Atari BASIC understood hexadecimal numbers and we all had eight fingers on each hand, there would be no need for this chapter. Instead, to use this book you have to make many conversions back and forth between hexadecimal (''hex'') and decimal notation. Many BASIC users have never had to learn that process.

Virtually all the references to addresses and other values in this book are given in hexadecimal notation (or simply ''hex'' to us insiders). For example, we learn that the Atari BASIC ROM cartridge has $A000 for its lowest address and that location $80 contains a pointer to BASIC's current LOMEM. But what does all that mean?

First of all, if you are not familiar with 6502 assembly language, let me point out that there is a convention that a number preceded by a dollar sign ($80) is a hexadecimal number, even if it contains only decimal digits. Also, notice that in the Source Listing all numbers in the first three columns are hexadecimal, even though the dollar sign is not present. (To the right of those columns, though, only those numbers preceded by a dollar sign are in hex.)

Now, suppose I wanted to look at the contents of location $A4AF (SNTAB in the listing). Realistically, the only way to look at a memory location from BASIC is via the PEEK function (and see the next chapter if you are not sure how to use PEEK in this situation). But BASIC's language syntax requires a decimal number with PEEK — for instance, PEEK (15).

Obviously, we need some way to convert from hexadecimal to decimal. Aside from going out and buying one of the calculators made just for this purpose, the best way is probably to let your computer help you. And the computer can help you
even if you only understand BASIC. As an example, here's a BASIC program that will convert hex to decimal notation:

```
10 DIM HEX$(23), NUM$(4)
20 HEX$ = "@ABCDEFGHI#######JKLMNO"
30 CVHEX = 9000
100 PRINT : PRINT "GIVE ME A HEX NUMBER ";
110 INPUT NUM$
120 GOSUB CVHEX
130 PRINT "HEX "; NUM$; " = DECIMAL "; NUM
140 GOTO 100
9000 REM THE CONVERT HEX TO DECIMAL ROUTINE
9010 NUM = 0
9020 FOR I = 1 TO LEN(NUM$)
9030 NUM = NUM * 16 + ASC(HEX$(ASC(NUM$(I)) - 47)) - 64
9040 NEXT I: RETURN
```

Now, while this program might be handy for a few purposes, it would be much neater if we could simply use its capabilities anytime we wanted to examine or change a location (or its contents) referred to by a hex address or data. And so shall it be used.

If we remove lines 100 through 140, inclusive, then any BASIC program which incorporates the rest of the program may change a hex number into decimal by simply

1. placing the ATASCII form of the hex number in the variable NUM$,
2. calling the convert routine at line 9000 (via GOSUB CVHEX), and
3. using the result, which is returned in the variable NUM.

In the next chapter, we will immediately begin to make use of this routine. If you are not used to hex notation, you might do well to type in and play with this program before proceeding.

Finally, before we leave this subject, let's examine a routine which will allow us to go the other way — that is, convert decimal to hex:

```
40 DIM DEC$(16): DEC$ = "0123456789 ABCDEF"
50 CVDEC = 9100
100 PRINT : PRINT "GIVE ME A DECIMAL NUMBER ";
```
Chapter One

110 INPUT DEC: NUM=DEC
120 GOSUB CVDEC: REM 'NUM' is destroyed by this
130 PRINT DEC; " Decimal = " ; NUM$ ; " Hex"
140 GOTO 100
9100 REM CONVERT DECIMAL TO HEX ROUTINE
9110 DIV=4096
9120 FOR I=1 TO 4
9130 N=INT(NUM/DIV): NUM$(I,I)=DEC$(N+1)
9140 NUM=NUM-DIV*N: DIV=DIV/16
9150 NEXT I
9160 RETURN

These lines are meant to be added to the previous program, though they can be used alone if you simply add this line:

10 DIM NUM$(4)

We will use portions of these programs in later chapters, but we may compress some of the code into fewer lines simply to save wear and tear on our fingers. If you study these routines, you’ll recognize them in their transformed versions.
Chapter Two

PEEKing and POKEing

In contrast to languages which include direct machine addressing capability, like "C" and Forth, and in contrast to "school" languages like Pascal and Fortran, which specifically prevent such addressing, BASIC provides a sort of halfway measure in machine accessibility.

POKE is a BASIC statement. Its syntax is POKE < address>, < data>. Naturally, both < address> and < data> may be constants, variables, or even full-blown expressions:

POKE 82,0: REM change left screen margin to zero produces the same result as

LEFTMARGIN = 82:POKE LEFTMARGIN,0

PEEK, on the other hand, is a BASIC function. It cannot stand alone as a statement. To use PEEK, we either PRINT the value (contents) of a PEEKed location, assign a PEEKed value to a variable, or test the value for some condition:

POKE 82, PEEK(82) + 1 : REM move the left margin in a space

PRINT PEEK(106) : REM where is the top of system memory?

IF PEEK(195) = 136 THEN PRINT "'End of File'

In the first example, the number POKEd into 82 will be whatever number was stored before, plus 1. As explained in Part 1, the PEEK function is executed before the POKE.

An aside: Just where did I get those addresses I used in the PEEKs and POKEs? One way to find them is to peruse the listings of Atari’s operating system, available in Atari’s technical manuals set, and the listing of BASIC in this book. Another way would be to use a book (like COMPUTE! Books’ Mapping the Atari) or a reference card designed specifically to tell you about such addresses.

And one more thing to consider before moving on. If we counted all of the bit patterns possible in a single 8-bit byte (like
Chapter Two

01010101, 11110000, and 00000001, where each 1 or 0 represents a single on or off bit, we would discover that there are 256 unique combinations, ranging in value from 0 to 255. Since each memory location can hold only one byte, it is not surprising to learn that the PEEK function will always return a number from 0 to 255 ($00 to $FF). Similarly, BASIC will only POKE a data value that is an integer from 0 to 255. In fact, BASIC will convert any data to be POKEd to an integer number, rounding off any fractional parts.

So far so good. But suppose we want to examine a location which is actually a two-byte word, such as the line number where the last TRAPped error occurred, stored starting at location $BA hex or 186 decimal. PEEK only lets us look at one byte at a time. How do we look at two bytes? Simple: one byte at a time.

In most cases, words in a 6502-based machine are stored in memory with the least significant byte stored first. This means that the second byte of each word is a count of the number of 256's there are in its value, and the first byte is the leftovers. (Or we can more properly say that the first byte contains "the word's value modulo 256." ) Confused? Let's try restating that.

In decimal arithmetic, we can count from 0 to 9 in a single digit. To go beyond 9, we have a convention that says the digit second from the right represents the number of 10's in the number, and so on.

If we consider bytes to be a computer's digits, which in many ways they are, and if we remember that each byte may represent any number from 0 to 255 (or $00 to $FF), then it is logical to say that the next byte is a count of the number of 256's in the number. The only thing illogical is that the higher byte comes after the lower byte (like reading 37 as "7 tens and 3 ones" instead of what we are used to).

Some examples might help:

<table>
<thead>
<tr>
<th>a 6502 word in memory</th>
<th>as written in assembler</th>
<th>think of it as</th>
<th>decimal value</th>
</tr>
</thead>
<tbody>
<tr>
<td>01 00</td>
<td>$0001</td>
<td>0*256 + 1</td>
<td>1</td>
</tr>
<tr>
<td>00 01</td>
<td>$0100</td>
<td>1*256 + 0</td>
<td>256</td>
</tr>
<tr>
<td>02 04</td>
<td>$0402</td>
<td>4*256 + 2</td>
<td>1026</td>
</tr>
<tr>
<td>FF FF</td>
<td>$FFFF</td>
<td>255*256 + 255</td>
<td>65535</td>
</tr>
</tbody>
</table>

So let's examine that error line location:

PRINT PEEK(186) + 256 * PEEK(187)
Do you see it? Since the second byte is a count of the number of 256's in the value, we must multiply it by 256 to calculate its true value.

Now, in the case of line numbers, it is well and good that we print out a decimal value, since that is how we are used to thinking of them. But suppose you wished to print out some of BASIC's tables? You might very well wish to see the hex representations. The program presented here allows you to specify a hex address. It then presents you with the contents of the byte and the word found at that address, in both decimal and hex form.

```
10 DIM HEX$(23), NUM$(4)
20 HEX$ = "@ABCDEFGHI#######JKLMNO"
30 CVHEX = 9000
40 DIM DEC$(16); DEC$ = "0123456789ABCDEF"
50 CVDEC = 9100
100 PRINT : PRINT "WHAT ADDRESS TO VIEW ";
110 INPUT NUM$ : PRINT
120 PRINT "Address "; NUM$; " contains:"
130 GOSUB CVHEX : ADDR = NUM
140 NUM = PEEK(ADDR) : GOSUB CVDEC
150 PRINT , "byte "; PEEK(ADDR); " = $"; NUM$(3)
160 WORD = PEEK(ADDR) + 256 * PEEK(ADDR + 1)
170 NUM = WORD : GOSUB CVDEC
180 PRINT , "word "; WORD; " = $"; NUM$
190 GOTO 100
9000 REM THE CONVERT HEX TO DECIMAL ROUTINE
9010 NUM = 0
9020 FOR I = 1 TO LEN(NUM$)
9030 NUM = NUM * 16 + ASC(HEX$(ASC(NUM$(I)) - 47)) - 64
9040 NEXT I: RETURN
9100 REM CONVERT DECIMAL TO HEX ROUTINE
9110 DIV = 4096
9120 FOR I = 1 TO 4
9130 N = INT(NUM / DIV) : NUM$(I, I) = DEC$(N + 1)
9140 NUM = NUM - DIV * N : DIV = DIV / 16
9150 NEXT I
9160 RETURN
```

You may have noticed that lines 10 through 50 and lines 9000 to the end are the same as those used in the example
programs in the last chapter. And did you see line 160, where we obtained the word value by multiplying by 256?

As the last point of this chapter, we need to discuss how to change a word value. Obviously, in Atari BASIC we can’t POKE both bytes of a word at once any more than we could retrieve both bytes at once (although BASIC A + can, by using the DPOKE statement and DPEEK function). So we must invent a mechanism to do a double POKE.

Given that the variable ADDR contains the address at which we wish to POKE a word, and given that the variable WORD contains the value (in decimal) of the desired word, the following code fragment will perform the double POKE:

```
POKE ADDR + 1, INT(WORD/256)
POKE ADDR, WORD – 256 * PEEK(ADDR + 1)
```

This is kind of sneaky code, but calculating the most significant byte and POKEing the value in byte location ADDR + 1 first allows us to also use it as a kind of temporary variable in calculating the least significant byte. By PEEKing the location that already holds the high-order byte, we can subtract it from the original value. The remainder is WORD modulo 256 — the low-order byte.

And that’s about it. Hopefully, if you were not familiar with PEEK and POKE before, you now at least will not approach their use with too much caution. Generally, PEEKs will never harm either your running program or the machine, but don’t be surprised if a stray POKE or two sends your computer off into never-never land. After all, you may have just told BASIC to start putting your program into ROM, or worse.

On the other hand, if you have removed your diskettes and turned off your cassette recorder, the worst that can happen from an erring POKE is that you’ll have to turn the power off and back on again. So have at it. Happy PEEKing and POKEing.
Chapter 3 of Part 1 described the layout of the Variable Name Table and the Variable Value Table. In particular, we read that the Variable Name Table was built in a very simple fashion: Each new variable name, as it is encountered upon program entry, is simply added to the end of the list of names. The most significant bit of the last character of the name is turned on, to signal the end of that name. The contents of VNTP point to the beginning of the list of names, and the content of VNTD is the address of the byte after the end of the list.

Now, what does all that mean? What does it imply that we can do? Briefly, it implies that we can look at BASIC's memory and find out what variable names are in current use. Here's a program that will do exactly that:

```
32700 QQ=128:PRINT QQ,
32710 FOR Q=PEEK(130)+256*PEEK(131) TO PEEK(132)+256*PEEK(133)-1
32720 IF PEEK(Q)<128 THEN PRINT CHR$(PEEK(Q));:NEXT Q:STOP
32730 PRINT CHR$(PEEK(Q)-128):QQ=QQ+1:PRINT QQ,:NEXT Q:STOP
```

Actually, this is not so much a program as it is a program fragment. It is intended that you will type NEW, type in the above fragment, and then LIST the fragment to a disk file (LIST "D:LVAR") or to a cassette (LIST "C:"). Then type NEW again and ENTER or LOAD the program whose variables you want to list. Finally, use ENTER to re-enter the fragment from disk (ENTER "D:LVAR") or cassette (ENTER "C:"). Then type GOTO 32700 to obtain your Variable Name Table listing.

Of course, if you had OPENed a channel to the printer (OPEN #1,8,0,"P:"), you could change the PRINTs to direct the listing to the printer (PRINT #1; CHR$(<expression>)).
Chapter Three

How does the fragment work? The reason for the start and end limits for the FOR loop are simple: word location 130 ($82) contains the pointer to the beginning of the Variable Name Table and word location 132 ($84) contains the pointer to the end of that same table, plus 1. So we simply traipse through that table, printing characters as we encounter them — except that when we encounter a character with its most significant bit on (IF PEEK(Q) > 127), we turn off that bit before printing it and start the next name on a new line.

Notice that we use the variable QQ to allow us to print out the token value for each variable name. We will use this information in some later chapters.

Also note that the variable names QQ and Q will appear in your variable name listing. Sorry. We can write a program which would accomplish the same thing without using variables, but it would be two or three times as big and much harder to understand. Of course, if you consistently use certain variable names, such as I and J in FOR-NEXT loops, you could use those names here instead, thus not affecting the count of variables in use.

Incidentally, the STOP at the end of the third line should be unnecessary, since the table is supposed to end with a character with its upper bit on. But I’ve learned not to take chances — things don’t always go as they’re supposed to.
Chapter Four

Variable Values

In this chapter, we will show how you can determine the value of any variable by inspecting the Variable Value Table. Actually, in many respects this is a waste of effort. After all, if I need to know the value of the variable TOTAL, I can just type PRINT TOTAL.

But this book is supposed to be a guide, and there are a few uses for this information, particularly in assembly language subroutines, and it is instructive in that it gives us an inkling of what BASIC goes through to evaluate a variable reference.

It will probably be better to present the program first, and then explain what it does. Before doing so, though, note that the program fragment expects you to give it a valid variable token (128 through 255). No checks are made on the validity of that number, since we are all intelligent humans here and since we want to save program space. Enough. The program:

```
32500 PRINT :PRINT "WHAT VARIABLE NUMBER "::INPUT Q
32505 Q=PEEK(134)+256*PEEK(135)+(Q-128)*8
32510 PRINT :PRINT "VARIABLE NUMBER ";PEEK(Q+1),
32515 ON INT(PEEK(Q)/64) GOTO 32600,32650
32520 PRINT "IS A NUMBER, ":PRINT "VALUE ";
32525 QEXP=PEEK(Q+2):IF QEXP>127 THEN PRINT "-";:QEXP=QEXP-128
32530 QNUM=0:FOR QQ=Q+3 TO Q+7
32535 QNUM=QNUM*100+PEEK(QQ)-6*INT(PEEK(QQ)/16):NEXT QQ
32540 QEXP=QEXP-68:IF QEXP=0 THEN 32555
32545 FOR QQ=QEXP TO SGN(QEXP) STEP -SGN(QEXP)
```
32550  QNUM=(QEXP>0)*QNUM*100+(QEXP<0)*QNUM/100: NEXT QQ
32555  PRINT QNUM: PRINT: GOTO 32500
32570  IF PEEK(Q)/2<>INT(PEEK(Q)/2) THEN 32580
32575  PRINT ,"AND IS NOT YET DIMENSIONED ": POP: GOTO 32500
32580  PRINT ,"ADDRESS IS "; PEEK(Q+2)+256*PEEK(Q+3): RETURN
32600  PRINT "IS AN ARRAY, ": GOSUB 32570
32610  PRINT ,"DIM 1 IS "; PEEK(Q+4)+256*PEEK(Q+5)
32615  PRINT ,"DIM 2 IS "; PEEK(Q+6)+256*PEEK(Q+7)
32620  GOTO 32500
32650  PRINT "IS A STRING, ": GOSUB 32570
32660  PRINT ,"LENGTH IS "; PEEK(Q+4)+256*PEEK(Q+5)
32665  PRINT ,"{3 SPACES}DIM IS "; PEEK(Q+6)+256*PEEK(Q+7)
32670  GOTO 32500

Did you get lost in all of that? I got lost several times as I wrote it, but it seems to work well. Shall we discuss it?

The first place where confusion may arise is when I ask you to give a variable token from 128 to 255, and then reveal that the entry in the Variable Value Table thinks variable numbers range from 0 to 127. Actually, there is no anomaly here. The variable token that you input is the token value of the variable in your program. The number in the table is its relative position. The numbers differ only in their uppermost bit.

The program uses the number you specify to form an address of an entry somewhere within the Variable Value Table. It then displays the internal variable number and examines the flag byte of the variable entry. Recall that the uppermost bit ($80, or 128) of the flag byte is on, if this variable is a string. The next bit ($40, or 64) is on if the variable is an array. If neither is on, the variable is a normal floating point number (or scalar, as it is sometimes called, to distinguish it from a floating point array). All this is decided and acted upon in line 32515.
Before examining what happens if the number is a scalar, let’s look at strings and arrays. Both start out (lines 32600 and 32650) by identifying themselves and calling a subroutine which determines if the variable has been DIMensioned yet. If not, the subroutine tells us so, removes the GOSUB entry from the stack, and starts the whole shebang over again. If the variable is DIMensioned, though, we print its address before returning. Note that the address printed is the relative address within the String/Array Table.

If the DIMension check subroutine returns, both string and array variables have their vitals printed out before the program asks you for another variable number. In the case of a string, we see the current length (as would be obtained by the LENgth function) and its dimension. For an array, we see both dimensions. Note that array dimensions here are always one greater than the user program specified, so that a zero dimension value means “this dimension is unused.”

Point of interest: this program will never print a zero for an array dimension. Why? Because Atari BASIC never places a zero in either dimension when the DIM statement is executed. In a way, this is a "feature" (a feature is a documented bug). It implies that we may code DIM XX(7) and yet use something like PRINT XX(N,O). In other words, a singly dimensioned array in Atari BASIC is exactly equivalent to a doubly dimensioned array with a 0 as the second subscript in the DIM statement.

Back to the listing. Fairly straightforward up until now. But look what happens if the variable is a scalar, a single floating point number.

First, we obtain the exponent byte; if its upper bit is on, the number is negative, so we print the minus sign before turning the bit off.

Second, we must loop through the five bytes of the mantissa, accumulating a value. The really strange part here is line 32535, so let’s examine it closely. As we get each byte, we must multiply what we have gotten so far by 100 (remember, floating point numbers are in BCD format, so each byte represents a power of 100). Then, what we really want to do is add in 10 times the higher digit in the byte, plus the lower digit. We could have gotten those numbers as follows:

\[ \text{NEWBCDVALUE} = \text{OLDBCDVALUE} \times 100 \]
\[ \text{HIGHER} = \text{INT} \left( \frac{\text{PEEK(QQ)}}{16} \right) \]
Chapter Four

\[ \text{LOWER} = \text{PEEK(QQ)} - 16 \times \text{HIGHER} \]
\[ \text{BYTEVALUE} = 10 \times \text{HIGHER} + \text{LOWER} \]
\[ \text{NEWBCDVALUE} = \text{NEWBCDVALUE} + \text{BYTEVALUE} \]
\[ \text{OLDBCDVALUE} = \text{NEWBCDVALUE} \]

Hopefully, your algebra is up to understanding how line 32535 is just a simplification of all that. If not, don't worry about it. It works.

But we still haven't accounted for the exponent. Now, exponents in the Atari floating point format are powers of 100 in "excess 64" notation, which simply means that you subtract 64 from the exponent to get the real power of 100. But wait! The implied decimal point is all the way to the left of the number. So we must bias our "excess 64" by the five multiplies-by-100 we did in deriving the BCD value. All that is done in line 32540.

Finally, we simply count the exponent down to one or up to minus one, depending on what it started at. And line 32545 is tricky, but not too much so. I will leave its inner workings as an exercise for you, the reader.

And, hard though it may be to believe, we arrive at line 32555 with the number in hand. Then we PRINT it.

Did we really have to go through all that? Not really, but perhaps it gives you an idea of what BASIC's GETTOK routine ($AB3E) does when it encounters a variable name.

Finally, to test all this out, you should type it in, LIST it to disk or cassette, use NEW, and then enter or load your favorite program. Finally, re-ENTER this program fragment from disk or cassette and type GOTO 32500. Just for fun, you might try finding the variable values for the following program:

```basic
10 A = 12.34567890 : B = 9876543210
20 C = 0.0000055677
30 GOTO 60
40 D$ = "WILL NEVER BE EXECUTED"
50 E(7) = 1
60 DIM F$(30), G$(40), H(9,17), J(7)
70 G$ = "ONLY THIS STRING WILL HAVE LENGTH"
```

Type this little guy in, ENTER the variable value printer, and RUN the whole thing. Answer the variable number prompt with numbers from 128 to 135 and see what you get. It's interesting!
Chapter Five

Examining the Statement Table

If you will recall, Chapter 3 in Part 1 discussed the various user tables that existed in Atari BASIC's RAM memory space. Specifically, it discussed the Variable Name Table, Variable Value Table, Statement Table, String/Array Table, and Runtime Stack.

In the last two chapters, we investigated the Variable Name Table and the Variable Value Table, showing how Atari BASIC can examine itself. So what is more logical than to now use Atari BASIC to display the contents of the Statement Table?

While we could write a program that would examine the tokenized program and produce source text, there is little incentive to do so. The task would be both very difficult and very redundant: BASIC's LIST command performs the same task very nicely, thank you.

What we can do, though, is write a program which will show the actual hex tokens used in a logical and almost readable form. Again, let's look at the program before decoding what it does.

10 DIM NUM$(4)
40 DIM DEC$(16):DEC$="0123456789ABCDEF"
50 CVDEC=9100
100 GOTO 32000
110 ERROR- THIS IS AN ERROR LINE
120 DATA AND, THIS, IS, DATA, 1,2,3
130 REM LINES 110 TO 130 ARE FOR DEMONSTRATION PURPOSES ONLY
9100 REM CONVERT DECIMAL TO HEX
9110 DIV=4096
9120 FOR I=1 TO 4
9130 N=INT(NUM/DIV):NUM$(I,I)=DEC$(N+1)
9140 NUM=NUM-DIV*N:DIV=DIV/16
9150 NEXT I
9160 RETURN
Chapter Five

32000  QQ=PEEK(136)+256*PEEK(137)
32010  Q=PEEK(QQ)+256*PEEK(QQ+1):QS=QQ:QQ
       =QQ+3
32015  IF Q>32767 THEN PRINT "--END--"; STOP
32020  QL=PEEK(QQ-1)+QS:PRINT "LINE NUMBER ";Q,"LINE LENGTH ";PEEK(QQ-1)
32030  QT=PEEK(QQ+1):PRINT "[2 SPACES]STM 
       T LENGTH ";PEEK(QQ),"STMT CODE ";PEEK(QQ+1)
32040  Q=PEEK(QQ)+QS:QQ=QQ+2
32050  IF QQ<Q THEN 32080
32060  IF Q<QL THEN PRINT :GOTO 32030
32070  PRINT :GOTO 32010
32080  IF QT>1 AND QT<55 THEN 32120
32090  PRINT "{2 SPACES}UNTOKENIZED::":QR=QS:QR=QR+1:IF Q
       Q<Q THEN 32100
32100  PRINT CHR$(PEEK(QR));:QQ=QQ+1:IF Q
       Q<Q THEN 32120
32110  PRINT :GOTO 32010
32120  NUM=PEEK(QQ):GOSUB CVDEC
32125  IF PEEK(QQ)>127 THEN PRINT " V=";NUM$(3):GOTO 32200
32130  IF PEEK(QQ)>15 THEN PRINT " ";NUM$(
       3);:GOTO 32200
32140  IF PEEK(QQ)=14 THEN GOTO 32170
32150  QQ=QQ+1:QN=PEEK(QQ):NUM=QN:GOSUB CVDEC
32155  PRINT ",NUM$(3);"="; ";IF QN=0 THEN 32200
32160  FOR QQ=QQ+1 TO QQ+QN-1:PRINT CHR$(
       PEEK(QQ));:NEXT QQ;GOTO 32190
32170  PRINT ", N=";
32180  FOR QQ=QQ+1 TO QQ+5:NUM=PEEK(QQ):G
       OSUB CVDEC:PRINT NUM$(3);:NEXT QQ
32190  QQ=QQ-1:PRINT
32200  QQ=QQ+1:IF QQ<Q THEN 32120
32210  PRINT :IF QQ<QL THEN 32030
32220  PRINT :GOTO 32010

Now, even if you don't want to type all that in, there are a few points to be made about it. First, note that lines 10 through 50 and 9100 through 9160 are the decimal-to-hex converter from
Chapter Five

Chapter 2. Then, let's start with line 32000 and do a functional description, with the line numbers denoting the portion we are examining.

32000. Decimal 136 is hex $88, the location of STMTAB, the pointer to the user's program space.

32010, 32020. In each line, the first two bytes are the line number; the next byte is the line length (actually, the offset to next line). Remember, line 32768 is actually the direct statement.

32030, 32040. Within a line, each statement begins with a statement length (the offset to the next statement from the beginning of the line) and a statement token.

32050-32070. Boundary conditions are checked for.

32080-32110. REM becomes statement token 0, DATA is token 1 and the error token is 55 ($37). All three of them simply store the user's input unchanged.

32120. Remember, any token with its upper bit on indicates a variable number token. They really don't need to be special cased in this program, but we do so for readability.

32130. Operator tokens have values of 16 to 127 ($10 to $7F).

32140-32160. For string constants (also called string literals), we simply print out the string length and its contents (the characters between the quote signs).

32170-32180. For numeric constants, we simply print the hex values of all six bytes.

32190-32200. Clean-up. We ensure that we return for all remaining tokens (if any) in each statement and for all remaining statements (if any) in each line.

Observe the FOR-NEXT loop controls in line 32180. Why QQ + 1 TO QQ + 5 if we want six values printed out? Ah, but this is a trick. Note that the loop termination value (QQ + 5) involves the loop variable (QQ). The problem is, though, that the loop variable is changed by the prior implied assignment (QQ = QQ + 1) when the assignment takes place — which is, of course, before the determination of the value of "QQ + 5" takes place.

In other words, by the time we are ready to evaluate QQ + 5, the variable QQ has already been changed from its original value to its new, loop controlling value (QQ + 1).

Quite possibly, the proper general solution to using a FOR loop's variable in its own termination (or STEP) values is to
assign it to a temporary variable, thusly:

\[ \text{QTEMP} = \text{QQ} : \text{FOR QQ} = \text{QTEMP} + 1 \text{ TO QTEMP} + 6 \]

Did you notice that line 32160 actually has the same problem? Notice that we solved it there by adding -1 to the termination value to compensate for the +1 in the initialization assignment.

One last comment before leaving the subject of strange FOR-NEXT loops. In Atari BASIC (and, indeed, in virtually all microcomputer BASICS), the termination (TO) value and the STEP value are determined when the FOR statement is first executed and are NOT changeable. Example:

\[
\begin{align*}
10 & \text{X} = 7 : \text{Y} = 2 \\
20 & \text{FOR I} = 1 \text{ TO X STEP Y} \\
30 & \text{X} = \text{X} + 1 \\
40 & \text{Y} = \text{Y} + \text{X} \\
50 & \text{NEXT I}
\end{align*}
\]

This FOR loop will execute exactly four times (I=1, 3, 5, and 7). The fact that X and Y change within the loop has no effect on the actual loop execution.
The Runtime Stack is the last of the user RAM tables that we will discuss in Part 2.

Perhaps you noticed that we left out a discussion of the String/Array Table in Part 2. The omission was on purpose: there seems little purpose in PEEKing the contents of this table when BASIC's PRINT statement does an admirable job of letting you see all variable values. However, if you are so inclined, you could use the general purpose memory PEEKer program of Chapter 2 to view any portion of any memory, including the String/Array Table.

On the other hand, looking at the Runtime Stack is kind of fun and enlightening. And the program we will present here might even find use on occasion. If you are having trouble tracing a program's flow, through various GOSUBs and/or FOR loops, simply drop in the routine below and GOSUB to it at an appropriate place in your program. It will print out a LIFO (Last In, First Out) listing of all active GOSUB calls and FOR-NEXT loop beginnings.

```plaintext
10 FOR J=1 TO 3
20 GOSUB 30
30 FOR K=1 TO 5
40 GOSUB 50
50 JUNK=7:FOR Q=1 TO 2:GOSUB 32400
32400 QQ=PEEK(144)+256*PEEK(145)
32410 IF QQ<=PEEK(142)+256*PEEK(143) THEN PRINT "--END OF STACK--":STOP
32420 PRINT "AT LINE ";PEEK(QQ-3)+256*PEEK(QQ-2);
32430 PRINT ", OFFSET ";PEEK(QQ-1);
32440 IF PEEK(QQ-4)=0 THEN PRINT ", GOSUB ":QQ=QQ-4:GOTO 32410
32450 PRINT ", FOR (";PEEK(QQ-4);")":QQ=QQ-16:GOTO 32410
```
The first thing you might notice about this little routine is that, in contrast to all the programs we have used so far, it examines its portion of user RAM backward. That is, it starts at the top (high address) of the Runtime Stack area and works downward toward the bottom.

Again, nothing surprising. If you will recall the description of entries on this stack (pages 18-19 and 133-34), you will remember that every entry, whether a GOSUB or FOR, has a four-byte header. And, while FOR statements also have twelve bytes of termination and step value added, the four bytes are always at the top of each entry — they are the last items put on the stack.

Thus, we start at the top of the stack and examine four bytes. If the type byte is zero, it is a GOSUB entry, and all we must do is display the line number and statement offset. If we remove the four-byte header by subtracting 4 from our stack pointer, we are ready to examine the next entry.

In the case of a FOR entry, we similarly display the line number and statement offset. However, each FOR entry also has a variable token associated with it, so we also display that token's value. With the variable name lister of Chapter 2, you can find out which variable is controlling this FOR loop. Finally, note that after displaying a FOR loop entry, we remove sixteen bytes (the four-byte header and the two six-byte floating point values) in preparation for the next entry.

Incidentally, lines 10 through 50 are present as examples only. Add lines 32400 to 32450 to your own programs and see where you've come from.
In the last chapter, we discussed the last of the tables in user RAM. Now we will see how and where BASIC stores its internal ROM-based tables.

As we noted in Chapter 5 of Part 1 (and viewed via the listing program of Chapter 5 in this Part), there are four kinds of tokens in an Atari BASIC program: (1) statement name tokens, (2) operator tokens, (3) variable tokens, and (4) constant tokens (string and numeric constants). Also, we learned in Part 1 how the tokenizing process works, converting the user's ATASCII source code into tokens. What we didn't learn, though, was exactly what token replaces what BASIC keyword.

In this chapter, we present a program which will list all of the fixed tokens (those in ROM). Actually, the program presents three listings, each consisting of a list of token values with their associated ATASCII strings. But wait a moment! Three listings? There are only two ROM-based tables — SNTAB and OPNTAB.

Yes, but it seems that this program is also capable of listing the Variable Name Table. Why list it again, when we did it so well in Chapter 3? Because we wanted to show you how BASIC itself does it. In many ways, this program emulates the functions of the SEARCH routine at address $A462 in the source listing. And, yes, BASIC uses a single routine to search all three of these same tables. You might want to examine BASIC's SEARCH routine at the same time you peruse this listing.

```
100 REM we make use of the general purpose
110 REM token lister three times:
200 PRINT :PRINT "A LIST OF VARIABLE TOKENS"
210 ADDR=PEEK(130)+256*PEEK(131)
220 SKIP=0:TOKEN=128:GOSUB 1000
300 PRINT :PRINT "A LIST OF STATEMENT TOKENS"
310 ADDR=42159:SKIP=2:TOKEN=0:GOSUB 1000
400 PRINT :PRINT "A LIST OF OPERATOR TOKENS"
410 ADDR=42979:SKIP=0:TOKEN=16:GOSUB 1000
420 STOP
1000 REM a general purpose token listing routine
```
Chapter Seven

1001 REM
1002 REM On entry to this routine, the following
1003 REM variables have meanings:
1004 REM ADDR = address of beginning of table
1005 REM SKIP = bytes per entry to skip
1006 REM TOKEN = starting token number
1007 REM
1100 ADDR=ADDR+SKIP:IF PEEK(ADDR)=0 THEN RETURN
1110 PRINT TOKEN,:TOKEN=_TOKEN+1
1120 IF PEEK(ADDR)>127 THEN 1140
1130 PRINT CHR$(PEEK(ADDR));:ADDR=ADDR+1:GOTO 1120
1140 PRINT CHR$(PEEK(ADDR)-128):ADDR=ADDR+1:GOTO 1100

The main routine is actually lines 1100 through 1140 (while lines 1000 through 1007 simply explain it all). It's actually fairly simple. Each table is assumed to consist of a fixed number of bytes followed by a variable number of ATASCII bytes, the last of which has its upper bit on.

In line 1100, we skip over the fixed bytes (if any) and check for the end of the table. After that, we simply print the token value followed by the name.

Worth examining, though, are lines 200 through 420, where we call the main subroutine. First, note that the Variable Name Table has no bytes to skip and is located via its zero-page pointer. Naturally, the first variable token value is 128.

Each entry in the Statement Name Table (SNTAB, at location $A4AF) has two leading bytes (actually, the two-byte address, minus 1, of the syntax table entry for this statement). Statement name token values begin at zero, and 42159 is the decimal address of SNTAB.

Finally, the smallest-numbered operator token is 16 decimal (except for string and numeric constants, which are special cased). There are no leading bytes in the Operator Name Table, and it starts at location 42979 decimal (OPNTAB, at $A7E3).
Chapter Eight

What Takes Precedence?

There was one other ROM-based table mentioned in Part 1 which deserves some attention here. You may recall that when an expression is executed, the execution operators are given particular precedences, so that in BASIC, $2 + 3 \times 4$ equals 14, not 20. Chapter 7 of Part 1 does a particularly thorough job of explaining the concepts of precedence.

The program presented in this chapter prints out all of BASIC’s operator tokens along with their token values and their dual precedence values. Actually, the program provides a visual readout of OPRTAB (Operator PRecedence TABle, at $\text{AC3F}$).

In each pair of precedence values listed, the first number is the go-onto-stack value and the second is the come-off-stack value.

100 PRINT "A LIST OF OPERATOR TOKENS"
110 PRINT " WITH THEIR PRECEDENCE TABLE VALUES"
220 SKIP=0:TOKEN=128:GOSUB 1000
1000 ADDR=42979:REM WHERE OP NAMES START
1010 TOKEN=16:REM LOWEST TOKEN VALUE
1020 REM NOW THE MAIN CODE LOOP
1100 IF PEEK(ADDR)=0 THEN STOP
1110 PRINT TOKEN,:PREC=PEEK(44095+TOKEN-16)
1120 PRINT INT(PREC/16);";PREC-16*INT(PREC/16),
1130 PREC=PEEK(ADDR):ADDR=ADDR+1
1140 IF PREC<128 THEN PRINT CHR$(PREC);:GOTO 1130
1150 PRINT CHR$(PREC-128):TOKEN=TOKEN+1:GOTO 1100

If you closely examined the program in the last chapter, you will note a striking similarity to this program, especially lines 1100 through 1150. Actually, the only thing we have really added is the precedence printout of line 1120.

And note the form of the PEEK in line 1110. Then look at the line of code at address $\text{AAF1}$ in the BASIC listing. Given
the limitations of dissimilar languages, the code is identical. This is more evidence that you really can use BASIC as a tool to diagnose itself.
Now that Atari BASIC stands revealed before you, what do you do with it? Many authors have, even without benefit of the listing in this book, either used or fooled BASIC in ways that we who designed it never dreamed of.

For example, consider what happens if you change BASIC’s STARP pointer ($8C) to be equal to its ENDSTAR value ($8E). Remember, BASIC’s SAVE command saves everything from the contents of VNTP to the contents of STARP (as documented in Chapter 10 of Part 1). So changing what is in STARP is tantamount to telling BASIC to SAVE more (or less) than what it normally would. Presto! We can now save the entire array and string space to disk or tape, also.

Is it useful? Here’s one program that is, using the concepts we learned in the previous chapters.

```
30000 PRINT :PRINT "WHAT VARIABLE NUMBER DO YOU":PRINT,"WISH TO FIND ";
30010 INPUT QV
30020 QA=PEEK(130)+256*PEEK(131):QN=128
30030 IF QN=QV THEN 30060
30040 IF PEEK(QA)<128 THEN QA=QA+1:GOTO 30040
30050 QN=QN+1:QA=QA+1:GOTO 30030
30060 IF PEEK(QA)<128 THEN PRINT CHR$(PEEK(QA));:QA=QA+1:GOTO 30060
30070 PRINT CHR$(PEEK(QA)-128);" IS THE VARIABLE"
30100 QA=PEEK(136)+256*PEEK(137)
30110 QN=PEEK(QA)+256*PEEK(QA+1):QL=PEEK (QA+2):QSV=QA:QA=QA+3
30120 IF QN>32767 THEN PRINT "--END--":END
30130 QS=PEEK(QA):QT=PEEK(QA+1):QA=QA+2: IF QT>1 AND QT<55 THEN 30150
```
Chapter Nine

30140 QA=QSV+QL: GOTO 30110
30150 IF PEEK(QA)=QV THEN PRINT "LINE ";
     QN: GOTO 30140
30160 IF PEEK(QA)>15 THEN 30200
30170 IF PEEK(QA)=14 THEN QA=QA+6: GOTO 30200
30180 QA=QA+PEEK(QA+1)+1
30200 QA=QA+1: IF QA<QSV+QS THEN 30150
30210 IF QA<QSV+QL THEN 30130
30220 GOTO 30110

What does it do? It finds all the places in your program that you used a particular variable. And how do you use it? Type it in, LIST it to disk or cassette, and clear the user memory via NEW. Now type, ENTER, or LOAD the program you wish to investigate (and then SAVE it, if you haven’t already done so). Finally, ENTER this program fragment from the disk or cassette where you LISTed it and type GOTO 30000.

Although the program asks you for a variable number (which you can get via the program of Chapter 3), it doesn’t really matter if you don’t know it. The program will print your chosen variable’s name before giving all the references. If you chose wrong, try again.

And how does it work? Somewhat like the program token lister of Chapter 5, except that here we are simply skipping everything but variable name references. First, though, we use a modified Variable Name Table lister (lines 30020 through 30070) to tell you what name you chose.

Then, we start at the beginning of the program (line 30100) and check each user line number (30110 and 30120). Within each line, we loop through, checking all statements (30130), skipping entirely all REMs, DATA lines, and lines with syntax errors (line 30140). If we find ourselves in an expression, we check for a matching variable token reference (line 30150) and print it if found, after which we skip the rest of the line. We also skip over numeric and string constants (lines 30170 and 30180). Finally, we check to see if we are at the end of the statement (30200) or the end of a line (30210 and 30220).

This is a fairly large program fragment, and it will prove most useful in very large programs, where you can’t remember, for example, how many places you are using the variable name LOOP. So you might want to try to leave room in memory for this aid; you may be very glad you did.

140
Part Three

Atari BASIC Source Code
Some Miscellaneous Equates

- $0001$ PATSZ EQU $\$1$ ; PATCH AREA SIZE
- $0020$ ZICB EQU $\$20$ ; zero PageIOCB
- $0080$ ZPG1 EQU $\$80$ ; beginning of BASIC's zero page
- $0480$ MISCR1 EQU $\$480$ ; syntax stack, etc.
- $0500$ MISCRAM EQU $\$500$ ; other RAM usage
- $E456$ CIO EQU $\$E456$ ; in OS ROMs
- $0340$ IOCBORG EQU $\$340$ ; where IOCBs start
- $0300$ DCBORG EQU $\$300$ ; where DCB (for SIO) is
- $A000$ ROM EQU $\$A000$ ; begin code here
- $0022$ ZFP EQU $\$02$ ; begin flag point work area
- $009B$ CR EQU $\$9B$ ; ATASCII end of line
- $02E7$ LMADR EQU $\$2E7$ ; system lo mem
- $02E5$ HMADR EQU $\$2E5$ ; system high mem
- $D800$ FPORG EQU $\$D800$ ; flag point in OS ROMs
- $0011$ BRKBYT EQU $\$11$ ; warmstart flag
- $0008$ WARMFL EQU $\$08$ ; get a random byte here
- $028A$ RNDLOC EQU $\$28A$ ; cartridge init vector
- $00FD$ CRTGI EQU $\$FDC$ ; the "?" for INPUT statement
- $E471$ BYELOC EQU $\$E471$ ; to go for BYE
- $008A$ DOSLOC EQU $\$8A$ ; via here to exit to DOS
- $0055$ SCRX EQU $\$55$ ; X AXIS
- $0054$ SCRY EQU $\$54$ ; Y AXIS
- $02C4$ CREGS EQU $\$2C4$ ; COLOR REGISTER
- $02FB$ SVCOLOR EQU $\$2FB$ ; SAVE COLOR FOR CIO
- $D208$ SREG1 EQU $\$D208$ ; SOUND REG 1
- $D200$ SREG2 EQU $\$D200$ ; SOUND REG 2
- $D201$ SREG3 EQU $\$D201$ ; SOUND REG 3
- $D20F$ SKCTL EQU $\$D20F$ ; sound control
- $0270$ GRFBAS EQU $\$270$ ; 1ST GRAPHICS FUNCTION ADDR
- $02FE$ DSPFLG EQU $\$2FE$ ; ATARI DISPLAY FLAG
- $003E$ APHM EQU $\$E$ ; APPLICATION HIGH MEM

Zero Page

RAM Table Pointers

<table>
<thead>
<tr>
<th>RAM Address</th>
<th>ORG</th>
<th>ZPG1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0000$</td>
<td>LOWMEM</td>
<td></td>
</tr>
<tr>
<td>$0009$</td>
<td>ARGOPS</td>
<td></td>
</tr>
<tr>
<td>$0030$</td>
<td>ARGSTK</td>
<td></td>
</tr>
<tr>
<td>$0080$</td>
<td>OUTBUFF</td>
<td>DS</td>
</tr>
<tr>
<td>$0082$</td>
<td>VNTP</td>
<td>DS</td>
</tr>
<tr>
<td>$0084$</td>
<td>VNTP</td>
<td>DS</td>
</tr>
<tr>
<td>$0086$</td>
<td>EVDVT</td>
<td>DS</td>
</tr>
<tr>
<td>$0088$</td>
<td>STMTAB</td>
<td>DS</td>
</tr>
<tr>
<td>$009A$</td>
<td>STMCUR</td>
<td>DS</td>
</tr>
<tr>
<td>$009C$</td>
<td>STARF</td>
<td>DS</td>
</tr>
<tr>
<td>$009E$</td>
<td>ENDDR</td>
<td>DS</td>
</tr>
<tr>
<td>$009F$</td>
<td>RUNSTK</td>
<td>DS</td>
</tr>
<tr>
<td>$0099$</td>
<td>TOPROM</td>
<td>DS</td>
</tr>
<tr>
<td>$009A$</td>
<td>MEMTOP</td>
<td>DS</td>
</tr>
<tr>
<td>$009B$</td>
<td>MEOLFLG</td>
<td>DS</td>
</tr>
<tr>
<td>$009C$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

143
<table>
<thead>
<tr>
<th>Source Code</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Miscellaneous Zero Page RAM</strong></td>
</tr>
<tr>
<td>USED FOR FREQUENTLY USED VALUES</td>
</tr>
<tr>
<td>TO DECREASE ROM SIZE AND INCREASE EXECUTION SPEED. ALSO USED FOR VARIOUS INDIRECT ADDRESS POINTERS.</td>
</tr>
<tr>
<td><strong>0094</strong> = <strong>0001</strong></td>
</tr>
<tr>
<td><strong>0095</strong> = <strong>0002</strong></td>
</tr>
<tr>
<td><strong>0097</strong> = <strong>0002</strong></td>
</tr>
<tr>
<td><strong>0097</strong> = <strong>0002</strong></td>
</tr>
<tr>
<td><strong>0099</strong> = <strong>0002</strong></td>
</tr>
<tr>
<td><strong>009B</strong> = <strong>0002</strong></td>
</tr>
<tr>
<td><strong>009D</strong> = <strong>0002</strong></td>
</tr>
<tr>
<td><strong>009D</strong> = <strong>0002</strong></td>
</tr>
<tr>
<td><strong>009F</strong> = <strong>0001</strong></td>
</tr>
<tr>
<td><strong>009F</strong> = <strong>0001</strong></td>
</tr>
<tr>
<td><strong>00A0</strong> = <strong>0002</strong></td>
</tr>
<tr>
<td><strong>00A2</strong> = <strong>0002</strong></td>
</tr>
<tr>
<td><strong>00A4</strong> = <strong>0002</strong></td>
</tr>
<tr>
<td><strong>00A6</strong> = <strong>0001</strong></td>
</tr>
<tr>
<td><strong>00A7</strong> = <strong>0001</strong></td>
</tr>
<tr>
<td><strong>00A8</strong> = <strong>0001</strong></td>
</tr>
<tr>
<td><strong>00A8</strong> = <strong>0001</strong></td>
</tr>
<tr>
<td><strong>00A9</strong> = <strong>0001</strong></td>
</tr>
<tr>
<td><strong>00A9</strong> = <strong>0001</strong></td>
</tr>
<tr>
<td><strong>00AA</strong> = <strong>0001</strong></td>
</tr>
<tr>
<td><strong>00AA</strong> = <strong>0001</strong></td>
</tr>
<tr>
<td><strong>00AB</strong> = <strong>0001</strong></td>
</tr>
<tr>
<td><strong>00AC</strong> = <strong>0001</strong></td>
</tr>
<tr>
<td><strong>00AD</strong> = <strong>0001</strong></td>
</tr>
<tr>
<td><strong>00AE</strong> = <strong>0001</strong></td>
</tr>
<tr>
<td><strong>00AF</strong> = <strong>0001</strong></td>
</tr>
<tr>
<td><strong>00AF</strong> = <strong>0001</strong></td>
</tr>
<tr>
<td><strong>00BF</strong> = <strong>0001</strong></td>
</tr>
<tr>
<td><strong>00BF</strong> = <strong>0001</strong></td>
</tr>
<tr>
<td><strong>00C0</strong> = <strong>0001</strong></td>
</tr>
<tr>
<td><strong>00C0</strong> = <strong>0001</strong></td>
</tr>
<tr>
<td><strong>00C0</strong> = <strong>0001</strong></td>
</tr>
<tr>
<td><strong>00C0</strong> = <strong>0001</strong></td>
</tr>
<tr>
<td><strong>00C0</strong> = <strong>0001</strong></td>
</tr>
<tr>
<td><strong>00C0</strong> = <strong>0001</strong></td>
</tr>
<tr>
<td><strong>00C0</strong> = <strong>0001</strong></td>
</tr>
<tr>
<td><strong>00C0</strong> = <strong>0001</strong></td>
</tr>
<tr>
<td><strong>00C0</strong> = <strong>0001</strong></td>
</tr>
<tr>
<td><strong>00C0</strong> = <strong>0001</strong></td>
</tr>
<tr>
<td><strong>00C0</strong> = <strong>0001</strong></td>
</tr>
<tr>
<td><strong>00C0</strong> = <strong>0001</strong></td>
</tr>
<tr>
<td><strong>00C0</strong> = <strong>0001</strong></td>
</tr>
<tr>
<td><strong>00C0</strong> = <strong>0001</strong></td>
</tr>
<tr>
<td><strong>00C0</strong> = <strong>0001</strong></td>
</tr>
<tr>
<td><strong>00C0</strong> = <strong>0001</strong></td>
</tr>
<tr>
<td><strong>00C0</strong> = <strong>0001</strong></td>
</tr>
<tr>
<td><strong>00C0</strong> = <strong>0001</strong></td>
</tr>
<tr>
<td><strong>00C0</strong> = <strong>0001</strong></td>
</tr>
<tr>
<td><strong>00C0</strong> = <strong>0001</strong></td>
</tr>
<tr>
<td><strong>00C0</strong> = <strong>0001</strong></td>
</tr>
<tr>
<td><strong>00C0</strong> = <strong>0001</strong></td>
</tr>
<tr>
<td><strong>00C0</strong> = <strong>0001</strong></td>
</tr>
<tr>
<td><strong>00C0</strong> = <strong>0001</strong></td>
</tr>
<tr>
<td><strong>00C0</strong> = <strong>0001</strong></td>
</tr>
<tr>
<td><strong>00C0</strong> = <strong>0001</strong></td>
</tr>
<tr>
<td><strong>00C0</strong> = <strong>0001</strong></td>
</tr>
<tr>
<td><strong>00C0</strong> = <strong>0001</strong></td>
</tr>
<tr>
<td><strong>00C0</strong> = <strong>0001</strong></td>
</tr>
<tr>
<td><strong>00C0</strong> = <strong>0001</strong></td>
</tr>
<tr>
<td><strong>00C0</strong> = <strong>0001</strong></td>
</tr>
<tr>
<td><strong>00C0</strong> = <strong>0001</strong></td>
</tr>
<tr>
<td><strong>00C0</strong> = <strong>0001</strong></td>
</tr>
</tbody>
</table>

144
Source Code

Argument Work Area (AWA)

Floating Point Work Area

\[ \begin{align*}
00CB & = 00D2 \quad \text{ORG} \quad ZFP \\
00D2 & = 0001 \quad \text{TVTYPE} \quad \text{DS} \quad 1 \quad \text{; VARIABLE TYPE} \\
00D3 & = 0001 \quad \text{TVNUM} \quad \text{DS} \quad 1 \quad \text{; VARIABLE NUMBER} \\
00D4 & = 0006 \quad \text{FPREC} \quad \text{EQU} \quad 6 \quad \text{; LENGTH OF FLOATING POINT} \\
00D5 & = 0005 \quad \text{FMPREC} \quad \text{EQU} \quad \text{FPREC-1} \quad \text{; MANTISSA} \\
00D6 & = 0001 \quad \text{FR0} \quad \text{DS} \quad 1 \quad \text{; FP REG0} \\
00D7 & = 0005 \quad \text{FR0M} \quad \text{DS} \quad \text{FPREC-1} \quad \text{; FP REG0 MANTISSA} \\
00D8 & = 0006 \quad \text{FRE} \quad \text{DS} \quad \text{FPREC} \quad \text{; FP REG0 EXP} \\
00D9 & = 0001 \quad \text{FR1} \quad \text{DS} \quad 1 \quad \text{; FP REG 1} \\
00DA & = 0005 \quad \text{FR1M} \quad \text{DS} \quad \text{FPREC-1} \quad \text{; FP REG1 MANTISSA} \\
00DB & = 0006 \quad \text{FR2} \quad \text{DS} \quad \text{FPREC} \quad \text{; FP REG 2} \\
00DC & = 0001 \quad \text{FRX} \quad \text{DS} \quad 1 \quad \text{; FP SPARE}
\end{align*} \]

RAM for ASCII to Floating Point Conversion

\[ \begin{align*}
00ED & = 0001 \quad \text{EEXP} \quad \text{DS} \quad 1 \quad \text{; VALUE OF E} \\
00EE & = 0001 \quad \text{FRSIGN} \quad \text{DS} \quad 1 \quad \text{; SIGN OF EXPONENT} \\
00EF & = 0001 \quad \text{NSIGN} \quad \text{DS} \quad 1 \quad \text{; SIGN OF \#} \\
00F0 & = 0001 \quad \text{SQRCTNT} \quad \text{DS} \quad 1 \quad \text{; SIGN OF EXPONENT} \\
00F1 & = 0001 \quad \text{PLYCNT} \quad \text{DS} \quad 1 \quad \text{; SIGN OF EXPONENT} \\
00F2 & = 0001 \quad \text{ESIGN} \quad \text{DS} \quad 1 \quad \text{; SIGN OF EXPONENT} \\
00F3 & = 0001 \quad \text{SMFLG} \quad \text{DS} \quad 1 \quad \text{; 1ST CHAR FLAG} \\
00F4 & = 0001 \quad \text{FCHRFLG} \quad \text{DS} \quad 1 \quad \text{; # OF DIGITS RIGHT OF DECIMAL}
\end{align*} \]

Input Buffer Controls

\[ \begin{align*}
00F2 & = 0001 \quad \text{CIX} \quad \text{DS} \quad 1 \quad \text{; CURRENT INPUT INDEX} \\
00F3 & = 0002 \quad \text{INBUFF} \quad \text{DS} \quad 2 \quad \text{; LINE INPUT BUFFER}
\end{align*} \]

Temps

\[ \begin{align*}
00F5 & = 0002 \quad \text{ZTEMP1} \quad \text{DS} \quad 2 \quad \text{; LOW LEVEL ZERO PageTEMPS} \\
00F7 & = 0002 \quad \text{ZTEMP4} \quad \text{DS} \quad 2 \\
00F9 & = 0002 \quad \text{ZTEMP3} \quad \text{DS} \quad 2
\end{align*} \]

Miscellany

\[ \begin{align*}
00FB & = 0001 \quad \text{DEGFLG} \quad \text{DS} \quad 1 \quad \text{; 0=RADIANS, 6= DEGREES} \\
00FD & = 0001 \quad \text{RADFLG} \quad \text{DS} \quad 1 \quad \text{; INDICATE RADIANS} \\
00FC & = 0002 \quad \text{FLPTR} \quad \text{DS} \quad 2 \quad \text{; POLYNOMIAL POINTERS} \\
00FE & = 0002 \quad \text{FPTR2} \quad \text{DS} \quad 2
\end{align*} \]

Miscellaneous Non-Zero Page RAM

\[ \begin{align*}
0100 & = 0480 \quad \text{ORG} \quad \text{MISCR1} \\
0480 & = 0480 \quad \text{STACK} \quad \text{EQU} \quad * \quad \text{; SYNTAX STACK} \\
0480 & = 0001 \quad \text{SIX} \quad \text{DS} \quad 1 \quad \text{; INPUT INDEX} \\
0481 & = 0001 \quad \text{SOX} \quad \text{DS} \quad 1 \quad \text{; OUTPUT INDEX} \\
0482 & = 0002 \quad \text{SPC} \quad \text{DS} \quad 2 \quad \text{; PGM COUNTER} \\
0484 & = 057E \quad \text{ORG} \quad \text{STACK+254} \\
057E & = 0001 \quad \text{LBPR1} \quad \text{DS} \quad 1 \quad \text{; LBUFF PREFIX 1} \\
057F & = 0001 \quad \text{LBPR2} \quad \text{DS} \quad 1 \quad \text{; LBUFF PREFIX 2} \\
0580 & = 0000 \quad \text{LBUFF} \quad \text{DS} \quad 128 \quad \text{; LINE BUFFER}
\end{align*} \]
**Source Code**

```assembly
06E8 = 05E8 ORG LBUFF+60
05E8 = 0006 PLYARG DS FPREC
05E6 = 0006 FPSCR DS FPREC
05EC = 0006 FPSCR1 DS FPREC
  = 05E6 FSCR EQU FPSCR
  = 05EC FPSCR1 EQU FPSCR1

**IOCB Area**

05F2 = 0340 ORG IOCBORG

**IOCB — I/O Control Block**

```assembly
; THERE ARE 8 I/O CONTROL BLOCKS
; 1 IOCB IS REQUIRED FOR EACH
; CURRENTLY OPEN DEVICE OR FILE.
;
0340 IOCB
0340 = 0001 ICHID DS 1 ; DEVICE HANDLER ID
0341 = 0001 ICNDO DS 1 ; DEVICE NUMBER
0342 = 0001 ICOMM DS 1 ; I/O COMMAND
0343 = 0001 ICSTA DS 1 ; I/O STATUS
0344 = 0001 ICBAL DS 1
0345 = 0001 ICBAH DS 1 ; BUFFERADR[H,L]
0346 = 0002 ICPUT DS 2 ; PUT A BYTE VIA THIS
0348 = 0001 ICBLL DS 1
0349 = 0001 ICBLH DS 1 ; BUFFER LENGTH[H,L]
034A = 0001 ICUX1 DS 1 ; AUXILIARY 1
034B = 0001 ICUX2 DS 1 ; AUXILIARY 2
034C = 0001 ICUX3 DS 1 ; AUXILIARY 3
034D = 0001 ICUX4 DS 1 ; AUXILIARY 4
034E = 0001 ICUX5 DS 1 ; AUXILIARY 5
0350 = 0070 DS ICLEN EQU *-IOCB
  = 010 ICLEN EQU ICLEN*7 ; SPACE FOR 7 MORE IOCBs

**ICCOM Value Equates**

```assembly
= 0001 ICOIN EQU $01 ; OPEN INPUT
= 0002 ICOOUT EQU $02 ; OPEN OUTPUT
= 0003 ICOIO EQU $03 ; OPEN UN/OUT
= 0004 ICBR EQU $04 ; GET BINARY RECORD
= 0005 ICGTR EQU $05 ; GET TEXT RECORDS
= 0006 ICGBC EQU $06 ; GET BINARY CHAR
= 0007 ICGTC EQU $07 ; GET TEXT CHAR
= 0008 ICPRR EQU $08 ; PUT BINARY RECORD
= 0009 ICPTR EQU $09 ; PUT TEXT RECORD
= 000A ICPCB EQU $0A ; PUT BINARY CHAR
= 000B ICPTC EQU $0B ; PUT TEXT CHAR
= 000C ICLOSE EQU $0C ; CLOSE FILE
= 000D ICSTAT EQU $0D ; GET STATUS
= 000E ICOFDC EQU $0E ; DEVICE DEPENDENT
= 000F ICMAX EQU $0F ; MAX VALUE
= 0010 ICFREE EQU $10 ; IOCB FREE INDICATOR
= 0011 ICGR EQU $11 ; OPEN GRAPHICS

**ICSTA Value Equates**

```assembly
= 0001 ICOK EQU $01 ; STATUSGOOD, NO ERRORS
= 0002 ICSTR EQU $02 ; TRUNCATED RECORD
= 0003 ICSEOF EQU $03 ; END OF FILE
= 0080 ICBRRK EQU $80 ; BREAK KEY ABORT
= 0081 ICSDNR EQU $81 ; DEVICE NOT READY
= 0082 ICSEND EQU $82 ; NON-EXISTENT DEVICE
= 0083 ICSDER EQU $83 ; DATA ERROR
= 0084 ICS1VC EQU $84 ; INVALID COMMAND
= 0085 ICSNIP EQU $85 ; DEVICE/FIELD NOT OPEN
= 0086 ICSPIN EQU $86 ; INVALID IOCB NUMBER
= 0087 ICSWPE EQU $87 ; WRITE PROTECTION
```

146
Equates for Variables

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>EVTYPE EQU 0</td>
</tr>
<tr>
<td>0000</td>
<td>EVSTR EQU $80</td>
</tr>
<tr>
<td>0004</td>
<td>EVARRAY EQU $40</td>
</tr>
<tr>
<td>0002</td>
<td>EVSDTA EQU $02</td>
</tr>
<tr>
<td>0001</td>
<td>EVDIM EQU $01</td>
</tr>
<tr>
<td>0000</td>
<td>EVSCALER EQU $00</td>
</tr>
<tr>
<td></td>
<td>EVNUM EQU 1</td>
</tr>
<tr>
<td>0002</td>
<td>EVVALUE EQU 2</td>
</tr>
<tr>
<td>0004</td>
<td>EVSADR EQU 4</td>
</tr>
<tr>
<td>0006</td>
<td>EVSDIM EQU 6</td>
</tr>
<tr>
<td>0002</td>
<td>EVAADR EQU 2</td>
</tr>
<tr>
<td>0004</td>
<td>EVAD1 EQU 4</td>
</tr>
<tr>
<td>0006</td>
<td>EVAD2 EQU 6</td>
</tr>
</tbody>
</table>

Equates for Run Stack

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0004</td>
<td>GHEAD EQU 4</td>
</tr>
<tr>
<td>000C</td>
<td>FBODY EQU 12</td>
</tr>
<tr>
<td>0001</td>
<td>GPLN EQU 3</td>
</tr>
<tr>
<td>0000</td>
<td>GTYPE EQU 0</td>
</tr>
<tr>
<td>0006</td>
<td>FSTEP EQU 6</td>
</tr>
<tr>
<td>0008</td>
<td>FLIM EQU 0</td>
</tr>
</tbody>
</table>

ROM Start

Cold Start

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>COLD START - REINITIALIZES ALL MEMORY</td>
</tr>
<tr>
<td></td>
<td>WIPES OUT ANY EXISTING PROGRAM</td>
</tr>
<tr>
<td>A000</td>
<td>COLDSTART</td>
</tr>
<tr>
<td>A000</td>
<td>A5CA LDA LOADFLG</td>
</tr>
<tr>
<td>A002</td>
<td>D004 D008 COLD1</td>
</tr>
<tr>
<td>A004</td>
<td>A5B0 LDA WARMFLG</td>
</tr>
<tr>
<td>A006</td>
<td>A045 A04D WARMSTART BNE WARMFLG</td>
</tr>
<tr>
<td>A008</td>
<td>COLD1</td>
</tr>
<tr>
<td>A008</td>
<td>A2PP LDX #$FF</td>
</tr>
<tr>
<td>A00A</td>
<td>9A TXS</td>
</tr>
<tr>
<td>A00B</td>
<td>D8 CLD</td>
</tr>
<tr>
<td>A00C</td>
<td>XNEW</td>
</tr>
<tr>
<td>A00C</td>
<td>AEE702 LDX LMADR</td>
</tr>
<tr>
<td>A00F</td>
<td>ACEB02 LDY LMADR+1</td>
</tr>
<tr>
<td>A012</td>
<td>86B0 STX LOMEM</td>
</tr>
<tr>
<td>A014</td>
<td>84B1 STY LOMEM+1</td>
</tr>
<tr>
<td>A016</td>
<td>9B08 LDA 40</td>
</tr>
<tr>
<td>A018</td>
<td>8592 STA MEOLFLG</td>
</tr>
<tr>
<td>A01A</td>
<td>85CA STA LOADFLG</td>
</tr>
<tr>
<td>A01C</td>
<td>CB INY</td>
</tr>
<tr>
<td>A01D</td>
<td>8A TXA</td>
</tr>
<tr>
<td>A01E</td>
<td>A282 LDX #VTTP</td>
</tr>
<tr>
<td>A020</td>
<td>9500 :CS1 STA 0,x</td>
</tr>
<tr>
<td>A022</td>
<td>E8 INX</td>
</tr>
<tr>
<td>A023</td>
<td>9480 STY 0,x</td>
</tr>
<tr>
<td>A025</td>
<td>E8 INX</td>
</tr>
<tr>
<td>A026</td>
<td>E092 CPX #MEMTOP+2</td>
</tr>
<tr>
<td>A028</td>
<td>90F6 A020 BCC :CS1</td>
</tr>
<tr>
<td>A02A</td>
<td>A286 LDX #VTTP</td>
</tr>
</tbody>
</table>

Cold Start - Reinitializes all memory, wipes out any existing program.
Source Code

A02C A001 LDY #1 ; FOR END OF VNT
A02E 207FAB JSR EXPLOW ; ZERO BYTE
A031 A28C LDX #255 ; EXPAND STMT TBL
A033 A003 LDY #3 ; BY 3 BYTES
A035 207FAB JSR EXPLOW ; GO DO IT

; A038 A900 LDA #0 ; SET 0
A03A A918 STA [VNTD],Y ; INTO VTDP
A03D 918A STA [STMCUR],Y ; INTO STMCUR+0
A03F C8 INY
A040 A980 LDA #$80 ; $80 INTO
A042 918A STA [STMCUR],Y ; STMCUR+1
A044 C8 INY
A045 A903 LDA #$03 ; $03 INTO
A047 918A STA [STMCUR],Y ; STMCUR+2

; A049 A90A LDA #10 ; SET PRINT TAB
A04B 85C9 STA PTABW ; WIDTH TO 10

Warm Start

; WARMSTART - BASIC RESTART
; DOES NOT DESTROY CURRENT PGM
A04D 60F88B JSR RUNINIT ; INIT FOR RUN
A056 2B418D SNX1JSR CLSALL ; GO CLOSE DEVICE 1-7
A053 2072BD SNX2JSR SETDZ ; SET E/L DEVICE 0
A056 A592 LDA MEOLFLG ; IF AN EOL INSERTED
A058 F003 ^A05D BEQ SNX3
A05A 2099BD JSR RSTSEOL ; THEN UN-INSERT IT
A05D 2057BD SNX3 JSR PREADY ; PRINT READY MESSAGE

Syntax

A060

Editor — Get Lines of Input

A060 SYNTAX
A068 A5CA LDA LOADFLG ; IF LOAD IN PROGRESS
A062 D09C ^A000 BNE COLDSTART ; GO DO COLDSTART
A064 A3F6 LDX #$FF ; RESTORE STACK
A066 9A TXS
A067 2051DA JSR INTLBF ; GO INT LBUFF
A06A A95D LDA #$EPCHAR
A06C 85C2 STA PROMPT
A06E 2092BA JSR GLGO ;
A071 20F4A9 JSR TSTBRK ; TEST BREAK
A074 D0EA ^A060 BNE SYNTAX ; BR IF BREAK

; A076 A900 LDA #0 ; INIT CURRENT
A078 B5F2 STA CIX ; INPUT INDEX TO ZERO
A07A B59F STA MAXCIX
A07C B594 STA COX ;OUTPUT INDEX TO ZERO
A07E B5A6 STA DIRFLG ;SET DIRECT SMT
A080 B5B3 STA SVONTX ; SET SAVE ONT CIX
A082 B5B0 STA SVONTC
A084 B5B1 STA SVVVTE ; VALUE IN CASE
A086 B5B4 LDA VNTD ; OF SYNTAX ERROR
A088 B5A0 STA SVVNTP
A08A B595 LDA VNTD+1
A08C B5AE STA SVVNTP+1

; A08E 20A1DB JSR SKBLANK ;SKIP BLANKS
A091 209FA1 JSR :GETLNUM ;CONVERT AND PUT IN BUFFER
A094 2B0B2A JSR :SETPCODE ; SET DUMMY FOR LINE LENGTH
A097 A5D5 LDA BININT+1
A099 1E02 ^A09D BPL :SYN0
A09B B5A6 STA DIRFLG
Source Code

A09D  20A1DB  JSR  SKBLANKS  ;  SKIP BLANKS
A09D  A4F2  LDY  CIX  ;  GET INDEX
A0A2  8A8  STY  STMSRTR  ;  SAVE IN CASE OF SYNTAX ERROR
A0A4  B1F3  LDA  [INBUFF],Y  ;  GET NEXT CHAR
A0A6  C99B  CMP  #CR  ;  IS IT CR
A0A8  ^A0B1  BNE  SYN1  ;  BR NOT CR
A0AA  2A46  BIT  DIRFLG  ;  IF NO LINE NO.
A0AC  3802  ^A060  BMI  SYNTAX  ;  THEN NO. DELETE
A0AE  4C89A1  JMP  :SDEL  ;  GO DELETE STMT
A0B1  ^SYN1
A0B1  ^XIF
A0B1  A594  LDA  COX  ;  SAVE COX
A0B3  85A7  STA  STMLBD  ;  AS PM TO STMT LENIUGH BYTE
A0B5  2C8CA2  JSR  :SETCODE  ;  DUMMY FOR STMT LENGTH
    ;
A0BB  20A1DB  JSR  SKBLANK  ;  GO SKIP BLANKS
A0BD  A0F4  LDY  #SNTAB&256  ;  SET UP FOR STMT
A0BF  A202  LDX  #2  ;  NAME SEARCH
A0C1  2B62A4  JSR  SEARCH  ;  AND DO IT
A0C4  B6F2  STX  CIX  ;  GET STMT NUMBER
A0CA  2C8CA2  JSR  :SETCODE  ;  GO SET CODE
A0CB  20A1DB  JSR  SKBLANK  ;  AND GO SYNTAX HIM
A0CD  935  ^A108  BCC  :SYNOK  ;  BR IF OK SYNTAX
A0CE  20C3A1  JSR  :SKBLANK  ;  ELSE SYNTAX ERROR
A0DF  A920  LDA  #S20  ;  BLANK IN IT'S PLACE
A0E1  0900  :SYN3A  ORA  #S80  ;  SET MAXCIX CHAR
A0E3  91F3  STA  [INBUFF],Y  ;  TO FLASH
A0E5  A49F  LDY  MAXCIX  ;  BR IF OK SYNTAX
A0E7  A94  LDA  #SNTAB&256  ;  NAME SEARCH
A0E9  85A6  STA  DIRFLG  ;  IN DIRFLG
A0EB  A4A8  LDX  STMSRTR  ;  RESTORE STMT START
A0ED  84F2  STX  CIX  ;  SET FOR FIRST STMT
A0EF  A203  LDX  #3  ;  SET FOR FIRST STMT
A0F1  86A7  STX  STMLBD  ;  INC TO CODE
A0F3  EB  INX  ;  AND SET COX
A0F4  8694  STY  COX  ;  AND SET COX
A0F6  A937  LDA  #CERR  ;  GARBAGE CODE
A0F8  2C8CA2  :SYN3  JSR  :SETCODE  ;  GO SET CODE
A0FB  :XDATA
A0FB  A4F2  LDY  CIX  ;  GET INDEX
A0FD  B1F3  LDA  [INBUFF],Y  ;  GET INDEX CHAR
A0FF  66F2  INC  CIX  ;  INC TO NXT
A101  C99B  CMP  #CR  ;  IS IT CR
A103  D0F3  ^A0P8  BNE  SYN3  ;  BR IF NOT
A105  2C8CA2  JSR  :SETCODE  ;
    ;
A108  A594  :SYNOK  LDA  COX  ;  GET DISPL TO END OF STMT
A10A  A4A7  LDY  STMLBD  ;  SET LENGTH BYTE
A10C  9180  STA  [OUTBUFF],Y  ;  SET LENGTH BYTE
A10E  A4F2  LDY  CIX  ;  GET INPUT DISPL
A110  88  DEY  ;  GET INPUT DISPL
A111  B1F3  LDA  [INBUFF],Y  ;  GET LAST CHAR
A113  C99B  CMP  #CR  ;  IS IT CR
A115  D09A  ^AB1  BNE  SYN1  ;  BR IF NOT
A117  A082  ^SYN4  LDY  #2  ;  SET LINE LENGTH
A119  A594  LDA  COX  ;  INTO STMT
Source Code

A11B 91B0 STA [OUTBUFF], Y
A11D 20A9 :SYN5 JSR GETSTMT ;GO GET STMT
A120 A880 LDA $0
A122 B003 'A127 BCS :SYN6
A124 :SYN5A
A124 20DDA9 JSR GETLL ;GO GET LINE LENGTH
A127 38 :SYN6 SEC
A128 E594 SBC COX ;ACU=LENGTH[OLD-NEW]
A12A F020 'A14C BFE :SYNIN ;BR NEW=OLD
A12C B013 'A141 BCS :SYNCON ;BR OLD+NEW
A12E 17FA8 JSR EXPLOW ;GO EXPAND
A130 68 PLA
A131 68 PLA
A132 A28A ILX #STMCUR ;POINT TO STMT CURRENT
A134 20FPA8 JSR EXPLOW ;GO EXPAND
A137 A597 LDA SVESA ;RESET STMCUR
A139 B5A8 STA STMCUR
A13B B5B8 LDA SVEA+1
A13D B5B8 STA STMCUR+1
A13F D00B 'A14C BNE :SYNIN
A141 48 :SYNCON PHA ;CONTRACT LENGTH
A142 20DD0A9 JSR GNXTL
A145 6B PLA
A146 A8 TAY
A147 A28A ILX #STMCUR ;POINT TO STMT CURRENT
A149 20FPA8 JSR CONTLOW ;GO CONTRACT
A14C A494 :SYNIN LDY COX ;STMT LENGTH
A14E 88 :SYN7 DEY ;MINUS ONE
A14F B180 LDA [OUTBUFF], Y ;GET BUFF CHAR
A151 91B8 STA [STMCUR], Y ;PUT INTO STMT TBL
A153 58 TYA
A154 D0F8 'A14E BNE :SYN7 ;BR IF NOT
A156 24A6 BIT DIRFLG ;TEST FOR SYNTAX ERROR
A158 502A 'A184 BVC :SYN8 ;BR IF NOT
A15A A51 LDA SVVTE
A15C ASLA
A15C +0A ASL A
A15D ASLA
A15D +0A ASL A
A15E ASLA
A15E +0A ASL A
A15F A8 TAY
A160 A28B ILX #ENDVVT
A162 20FPA8 JSR CONTLOW
A165 38 SEC
A166 A584 LDA VNNT ;CONTRACT VNT
A168 E5A8 SBC SVNTP
A16A A8 TAY
A16B A585 LDA VNNT+1
A16D E5AE SBC SVNTP+1
A16F A284 ILX #VNNT
A171 20FDA8 JSR CONTRACT
A174 24A6 BIT DIRFLG ;IF STMT NOT DIRECT
A176 1006 'A17E BPL :SYN9A ;THE BRANCH
A178 2078B8 JSR LDLINE ;ELSE LIST DIRECT LINE
A178 4C60A0 JMP SYNTAX ;THEN BACK TO SYNTAX
A17E 205CB5 :SYN9A JSR LLINE ;LIST ENTIRE LINE
A181 4C60A0 :SYN9 JMP SYNTAX
A184 10FB 'A181 :SYN8 BPL :SYN9
A186 4C5FA9 JMP EXECNL ;GO TO PROGRAM EXECUTOR
A189 20A2A9 :SDEL JSR GETSTMT ;GO GET LINE
A18C B0F3 'A181 BCS :SYN9 ;BR NOT FOUND
A18E 25DDA9 JSR GETLL ;GO GET LINE LENGTH
A191 48 PHA
Source Code

A192 20D8A9    JSR    GNXTL
A195 68    PLA
A196 A8    TAY
A197 A28A    LDX    #$STMCUR    ;GET STMCUR DISPL
A199 20FBA8    JSR    CONTLOW    ;GO DELETE
A19C 4C6A0    JMP    SYNTAX    ;GO FOR NEXT LINE

Get a Line Number

;GETLNUM-GET A LINE NO FROM ASCLT IN INBUFF
;TO BINARY INTO OUTBUFF

A19F :GETLNUM
A19F 2000D8    JSR    CVAFP    ;GO CONVERT LINE #
A1A2 9008    "A1AC    BCC    :GLNUM    ;BR IF GOOD LINE #
A1A4 :GLNUM

A1A4 A900    LDA    #0    ;SET LINE #
A1A6 85F2    STA    CIX
A1A8 A800    LDY    #$80    ;=$8000
A1AA 3009    "A1B5    BMI    :SLNUM

A1AC 2056AD    :GLNUM    JSR    CVFPI    ;CONVERT FP TO INT
A1AF 4AD5    LDY    BININT+1    ;LOAD RESULT
A1B1 30F1    "A1A4    BMI    :GLN1    ;BR IF LNO>32767
A1B3 A5D4    LDA    BININT

A1B5 :SLNUM
A1B5 84A1    STY    TSLNUM+1    ;SET LINE # HIGH
A1B7 85A0    STA    TSLNUM    ;AND LOW
A1B9 20CBA2    JSR    :SETCODE    ;OUTPUT LOW
A1BC A5A1    LDA    TSLNUM+1    ;OUTPUT HI
A1BE 85D5    STA    BININT+1
A1C0 4C08A2    JMP    :SETCODE    ;AND RETURN

SYNENT

;PERFORM LINE PRE-COMPILE

A1C3 :SYNENT
A1C3 A001    LDY    #1    ;GET PC HIGH
A1C5 B195    LDA    [SRCADR],Y    ;SET PGM COUNTERS
A1C7 859E    STA    CPC+1
A1C9 8DB304    STA    SPC+1
A1CC B8    DEY
A1CD B195    LDA    [SRCADR],Y
A1CF 859D    STA    CPC
A1D1 8DB204    STA    SPC
A1D4 A900    LDA    #0    ;SET STKLUL
A1D6 85A9    STA    STKLVL    ;SET STKLUL
A1DA A594    STA    COX    ;MOVE
A1DA 8DB104    STA    SOX    ;COX TO SOX
A1DD A5F2    LDA    CIX    ;MOVE
A1DF 8DB004    STA    SIX    ;CIX TO SIX

NEXT

;GET NEXT SYNTAX CODE
;AS LONG AS NOT FAILING

= A1E2    NEXT    EQU    *    ;GET NEXT CODE
A1E2 2A1A2    JSR    :NXSC
A1E5 301A    "A201    BMI    :ERNTV    ;BR IF REL-NON-TERMINAL
A1E7 C901    CMP    #1    ;TEST CODE=1
A1E9 902A    "A215    BCC    :GETADR    ;BR CODE=0 [ABS-NON-TERMINAL]
A1EB D080    "A1F5    BNE    :TSTSUCC    ;BR CODE >1
A1ED 2015A2    JSR    :GETADR    ;CODE=1 [EXTERNAL SUBROUTINE]
A1F0 90F0    "A1E2    BCC    :NEXT    ;BR IF SUB REPORTS SUCCESS
A1F2 4C6CA2    JMP    :FAIL    ;ELSE GO TO FAIL CODE
A1F5 C905    :TSTSUCC    CMP    #5    ;TEST CODE = 5

151
Source Code

A1F9 20A9A2 JSR :TERMTST  ; CODE>5 GO TEST TERMINAL
A1FC 9BE4 "A1E2 BCC :NEXT  ; BR IF SUCCESS
A1FE 4C6CA2 JMP :FAIL  ; ELSE GO TO FAIL CODE
 ;
A201 3B :ERNTV SEC  ; RELATIVE NON TERMINAL
A202 A200 LDX #0  ; TOKEN MINUS
A20A B002 "A20A BCS :ERNL1  ; BR IF RESULT PLUS
A20A A2FF LDX #$FF  ; ADD A MINUS
A20A 18 :ERNL1 CLC  ; RESULT PLUS CPC
A20B 659D ADC CPC  ; IS NEW CPC-1
A20D 4B PHA  ; SAVE CPC TO STACK
A20E 8A TXA  ; RETURN CONTROL TO TOP OF STACK
A211 4B PHA  ; EXECUTE NEXT SYNTAX CODE
A212 4C28A2 JMP :PUSH  ; GO PUSH
 = A215 :GETADR EQU *  ; GET DOUBLE BYTE ADR [-1]
A215 20A1A2 JSR :NXSC  ; GET NEXT CODE
A218 4B PHA  ; SAVE ON STACK
A219 20A1A2 JSR :NXSC  ; GET NEXT CODE
A21C 4B PHA  ; SAVE ON STACK
A21D 9009 "A22B BCC :PUSH  ; BR IF CODE =0
A21F 6B PLA  ; EXCHANGE TOP
A220 A8 TAY  ; 2 ENTRIES ON
A221 6B PLA  ; CPU STACK
A222 AA TXA  ;
A223 9B TYA  ;
A224 4B PHA  ;
A225 8A TXA  ;
A226 4B PHA  ;
A227 60 RTS  ; ELSE GOTO EXTERNAL SRT VIA RTS
 ;
PUSH  ;  PUSH TO NEXT STACK LEVEL
 ;
 = A228 :PUSH EQU *
A228 A6A9 LDX STKLVL  ; GET STACK LEVEL
A22A EE INX  ; PLUS 4
A22B E8 INX  ;
A22C E8 INX  ;
A22D E8 INX  ;
A22E F01F "A24F BCD :SSTB  ;BR STACK TOO BIG
A230 86A9 STX STKLVL  ; SAVE NEW STACK LEVEL
 ;
A232 A5F2 LDA CIX  ; CIX TO
A234 99804 STA SIX,X  ; STACK IX
A237 A594 LDA COX  ; COX TO
A239 998104 STA SOX,X  ; STACK OX
A23C A59D LDA CPC  ; CPC TO
A23E 998204 STA SPC,X  ; STACK CPC
A241 A59E LDA CPC+1  ;
A243 998304 STA SPC+1,X  ;
 ;
A246 6B PLA  ; MOVE STACKED
A247 859D STA CPC+1  ; PC TO CPC
A249 6B PLA  ;
A24A 859D STA CPC  ;
A24C 4CE2A1 JMP :NEXT  ; GO FOR NEXT
 ;
A24F 4C24B9 :SSTB JMP ERLTL  ;
 ;
POP  ;  LOAD CPC FROM STACK PC
 ;
 = A252 :POP EQU *
A252 A6A9 LDX STKLVL  ; GET STACK LEVEL
A254 D001 "A257 BNE :POPL  ; BR NOT TOP OF STACK
 ;
RTS

; TO SYNTAX CALLER
LDA SPC,X

; MOVE STACK PC
STA CPC

; TO CURRENT PC
LDA SPC+1,X
STA CPC+1

; X=X-4
DEX
OEX
OEX
STX STKLVL

; BR IF CALLER FAILING
BCS :FAIL
JMP :NEXT ; ELSE GO TO NEXT

FAIL

TERMINAL FAILED
LOOK FOR ALTERNATIVE [OR] OR
A RETURN INDICATOR

= A26C

; GET NEXT CODE
JSR :NXSC

; BR IF RNTV
BMI :FAIL

; TEST CODE = 2
CMP #2
TSTOR

; BR IF POSSIBLE OR
CODE = 0 OR 1

; INC PC BY TWO
BCS :FAIL

; AND CONTINUE FAIL PROCESS
:FAIL AND CONTINUE FAIL PROCESS

; TEST CODE=3
CMP #3
BEQ

; BR CODE =3 [RETURN] ; CODE>3 [RNTV] CONTINUE
INCPC

; IF THIS CIX
LDA CIX

; IS A NEW MAX
CMP MAXCIX

; THEN SET NEW MAX
BCC :SCIX

; CODE=2 [OR]
STA MAXCIX

; MOVE STACK INDEXES
LDA SIX,X

; TO CURRENT INDEXES
STA CIX

; TRY FOR SUCCESS HERE
STA SOX,X

; TRY FOR SUCCESS HERE
STA COX

; TRY FOR SUCCESS HERE
JMP :NEXT

Increment CPC

; INCCPC = INC CPC BY ONE

= A29A

; INCCPC EQU *
INC CPC

; ICPCR
BNE :ICPCR

; RC
INC CPC+1

NXSC

; GET NEXT SYNTAX CODE

= A268

; RX
INC CPC

; INC PC
JSR :INCCPC

; GET NEXT CODE
LDX #0

; GET NEXT CODE
LDA [CPC,X]

; RETURN
RTS
TERMSTT
; TEST A TERMINAL CODE
; A2A9 :TERMSTT
A2A9 C90F CMP #$8F ; TEST CODE=F
A2AB F00D A2BA BEQ :ECHNG ; BR CODE < F
A2AD B037 A2E6 BCS :SRCONT ; BR CODE > F
A2AF 68 PLA
A2B0 68 PLA
A2B1 A90C LDA #$EXP-1&255 ; PUSH EXP ADR
A2B3 48 PHA
A2B4 A9A6 LDA #$EXP/256 ; EXP ANTV CALL
A2B6 48 PHA
A2B7 4C2B A2 JMP ; PUSH ; GO PUSH

ECHNG
; EXTERNAL CODE TO CHANGE COX -1
; A2BA :ECHNG
A2BA 209A A2 JSR :INCCPC ; INC PC TO CODE
A2BD A000 LDY #0
A2BF B19D LDA [CPC], Y ; GET CODE
A2C1 A494 LDY COX ; GET COX
A2C3 88 DEY
A2C4 9180 STA [OUTBUFF], Y ; SET NEW CODE
A2C6 18 CLC
A2C7 60 RTS ; RETURN

SETCODE
; SET CODE IN ACV AT COX AND INC COX
; A2C8 :SETCODE
A2C9 A494 LDY COX ; GET COX
A2CA 9180 STA [OUTBUFF], Y ; SET CHAR
A2CC 696A INC COX ; INC COX
A2CE 8090 STA [OUTBUFF], Y ; SET NEW CODE
A2D0 60 RTS ; DONE
A2D1 4C2B9 JSR ; COVF JMP ; RLTL ; GO TO LINE TOO LONG ERR

Exits for IF and REM
A2D4 A2FF :EIF LDX #$FF ; RESET STACK
A2D6 9A TXS
A2D7 A594 LDA COX ; SET STMT LENGTH
A2D9 A4A7 LDY STMLBD
A2DB 9180 STA [OUTBUFF], Y
A2DD 4CB1A0 JMP ; XIF ; GO CONTINUE IF
A2E0 :EREM
A2E0 :BDATA
A2E2 9A TXS
A2E3 4C2BA0 JMP ; XDATA ; GO CONTINUE DATA

SRCONT
; SEARCH OF NAME TABLE AND TEST RESULT
; A2E6 :SRCONT
A2E6 20A1DB JSR SKPBLANK ; SKIP BLANKS
A2E9 A5F2 LDA CIX ; GET CURRENT INPUT INDEX
A2EB 58B3 CMP SVONTX ; COMPARE WITH SAVED IX
A2ED F016 A305 BEQ :SONT1 ; BR IF SAVED IX SAME
A2EF 85B3 STA SVONTX ; SAVE NEW IX
A2F1 A9A7 LDA #OPNTAB/256 ; SET UP FOR ONT
A2F3 A0E3 LDA #OPNTAB&255 ; SEARCH
A2F5 A2B0 LDX #0
A2F7 2B62A4 JSR SEARCH ; GO SEARCH
A2FA B028 A324 BCS :SONF ; BR NOT FOUND
A2FC 86B2 STX SVONTL ; SAVE NEW CIX
A2FE 18 CLC
A2FF A5AF LDA STENUM ADD $10 TO
A301 6910 ADC #$10 ; ENTRY NUMBER TO
A303 B5B0 STA SVONTC ; GET OPERATOR CODE
A305 A800 :SONT1 LDY #0 ; GET SYNTAX REQ CODE
A307 B19D LDA [CPC],Y ; DOES IT MATCH THE FOUND
A309 C5B0 CMP SVONTC ; BR IF MATCH
A30B F00E A31B BEQ :SONT2 ; WAS REQ NFNP
A30D C944 CMP #CNFNP ; BR IF NOT
A30F D006 "1'.324 BNE :SONTF ; GET WHAT WE GOT
A311 A5B0 LDA SVONTT GET OPERATOR CODE ; BR IF IT IS
A313 C944 CMP #CNFNP ; IS IT NFNA
A315 B002 A319 BCS :SONTS ; BR IF IT IS
A317 :SONTF
A317 38 SEC
A318 60 RTS
A319 A5B0 :SONTS LDA SVONTC GET REAL CODE
A31B 20CBA2 :SONT2 JSR :SETCODE ; GO SET CODE
A31E A6B2 LDX SVONTL ; INC CIX BY
A320 B6F2 STX CIX
A322 18 CLC
A323 60 RTS
A324 A900 :SONF, LDA #0 ; SET ZERO AS
A326 B5B0 STA SVONTC ; SAVED CODE
A328 38 SEC
A329 60 RTS
A32A A900 :TNVAR LDA #0 ; SET NUMERIC TEST
A32C F002 A330 BEQ :TVAR ;
A32E A900 :TSVAR LDA #$80 ; SET STR TEST
A330 85D2 :TVAR STA TVTYPE ; SAVE TEST TYPE
A332 28A1DB JSR SKPBLANK ; SKIP LEADING BLANKS
A335 A5F2 LDA CIX ; GET INDEX
A337 85AC STA TVSCIX FOR SAVING
A339 20FAA3 JSR :TSTALPH ; GO TEST FIRST CHAR
A33B 28EA2 JSR :SRCONT ; IF THIS IS A
A341 A5B0 LDA SVONTC ; RESVD NAME
A343 F000 A34D BEQ :TV1 ; BR NOT RSVPNAME
A345 A4B2 LDY SVONTL ; IF NEXT CHAR AFTER
A347 B1F3 LDA [INBUFF],Y ; RESERVED NAME
A349 C930 CMP #$30 ; NOT ALARM NUMERIC
A34B 9016 A363 BCC :TVFAIL ; THEN ERROR
A34D E6F2 :TV5 INC CIX ; INC TO NEXT CHAR
A34F 28FFE3 JSR :TSTALPH ; TEST ALPHANUM
A352 98F9 A34D BCC :TV1 ; BR IF ALPHA
A354 20AFDB JSR :TSTNUM ; TRY NUMBER
A357 98F4 A34D BCC :TV1 ; BR IF NUMBER
A359 B1F3 LDA [INBUFF],Y ; GET OFFENDING CHAR
A35B C924 CMP #$'S' ; IS IT $?
A35D F00E A365 BEQ :TVSTR ; BR IF $ [STRING]
A35E 20F2 A36D BIT TVTYPE ; THIS A SVAR SEARCH
A361 1009 A36C BPL :TVOK ; BR 'IF SVAR
A363 38 :TVFAIL SEC
A364 60 RTS
A365 24D2 :TVSTR BIT TVTYPE ; TEST SVAR SEARCH
A367 10F2 A363 BPL :TVFAIL ; BR IF SVAR

TVAR ; EXTERNAL SUBROUTINE FOR TNVAR & TSVAR

A32A A900 :TNVAR LDA #0 ; SET NUMERIC TEST
A32C F002 A330 BEQ :TVAR ;
A32E A900 :TSVAR LDA #$80 ; SET STR TEST
A330 85D2 :TVAR STA TVTYPE ; SAVE TEST TYPE
A332 28A1DB JSR SKPBLANK ; SKIP LEADING BLANKS
A335 A5F2 LDA CIX ; GET INDEX
A337 85AC STA TVSCIX FOR SAVING
A339 20FAA3 JSR :TSTALPH ; GO TEST FIRST CHAR
A33B 28EA2 JSR :SRCONT ; IF THIS IS A
A341 A5B0 LDA SVONTC ; RESVD NAME
A343 F000 A34D BEQ :TV1 ; BR NOT RSVPNAME
A345 A4B2 LDY SVONTL ; IF NEXT CHAR AFTER
A347 B1F3 LDA [INBUFF],Y ; RESERVED NAME
A349 C930 CMP #$30 ; NOT ALARM NUMERIC
A34B 9016 A363 BCC :TVFAIL ; THEN ERROR
A34D E6F2 :TV5 INC CIX ; INC TO NEXT CHAR
A34F 28FFE3 JSR :TSTALPH ; TEST ALPHANUM
A352 98F9 A34D BCC :TV1 ; BR IF ALPHA
A354 20AFDB JSR :TSTNUM ; TRY NUMBER
A357 98F4 A34D BCC :TV1 ; BR IF NUMBER
A359 B1F3 LDA [INBUFF],Y ; GET OFFENDING CHAR
A35B C924 CMP #$'S' ; IS IT $?
A35D F00E A365 BEQ :TVSTR ; BR IF $ [STRING]
A35E 20F2 A36D BIT TVTYPE ; THIS A SVAR SEARCH
A361 1009 A36C BPL :TVOK ; BR 'IF SVAR
A363 38 :TVFAIL SEC
A364 60 RTS
A365 24D2 :TVSTR BIT TVTYPE ; TEST SVAR SEARCH
A367 10F2 A363 BPL :TVFAIL ; BR IF SVAR

TVAR

A32A A900 :TNVAR LDA #0 ; SET NUMERIC TEST
A32C F002 A330 BEQ :TVAR ;
A32E A900 :TSVAR LDA #$80 ; SET STR TEST
A330 85D2 :TVAR STA TVTYPE ; SAVE TEST TYPE
A332 28A1DB JSR SKPBLANK ; SKIP LEADING BLANKS
A335 A5F2 LDA CIX ; GET INDEX
A337 85AC STA TVSCIX FOR SAVING
A339 20FAA3 JSR :TSTALPH ; GO TEST FIRST CHAR
A33B 28EA2 JSR :SRCONT ; IF THIS IS A
A341 A5B0 LDA SVONTC ; RESVD NAME
A343 F000 A34D BEQ :TV1 ; BR NOT RSVPNAME
A345 A4B2 LDY SVONTL ; IF NEXT CHAR AFTER
A347 B1F3 LDA [INBUFF],Y ; RESERVED NAME
A349 C930 CMP #$30 ; NOT ALARM NUMERIC
A34B 9016 A363 BCC :TVFAIL ; THEN ERROR
A34D E6F2 :TV5 INC CIX ; INC TO NEXT CHAR
A34F 28FFE3 JSR :TSTALPH ; TEST ALPHANUM
A352 98F9 A34D BCC :TV1 ; BR IF ALPHA
A354 20AFDB JSR :TSTNUM ; TRY NUMBER
A357 98F4 A34D BCC :TV1 ; BR IF NUMBER
A359 B1F3 LDA [INBUFF],Y ; GET OFFENDING CHAR
A35B C924 CMP #$'S' ; IS IT $?
A35D F00E A365 BEQ :TVSTR ; BR IF $ [STRING]
A35E 20F2 A36D BIT TVTYPE ; THIS A SVAR SEARCH
A361 1009 A36C BPL :TVOK ; BR 'IF SVAR
A363 38 :TVFAIL SEC
A364 60 RTS
A365 24D2 :TVSTR BIT TVTYPE ; TEST SVAR SEARCH
A367 10F2 A363 BPL :TVFAIL ; BR IF SVAR
A369  C8  INY ; INC OVER $  
A36A  D80D  "A379  BNE  :TVOK2 ; BR ALWAYS  
        ;  
A36C  B1F3  :TVOK  LDA  [INBUFF],Y ; GET NEXT CHAR  
A36E  C92B  CMP  #',' ; IS IT PAREN  
A370  D807  "A379  BNE  :TVOK2 ; BR NOT PAREN  
A372  C8  INY ; INC OVER PAREN  
A373  A940  LDA  #$40 ; OR IN ARRAY  
A375  05D2  ORA  TVTYPE ; CODE TO TVTYPE  
A377  85D2  STA  TVTYPE ;  
A379  A5AC  :TVOK2  LDA  TVSCIX ; GET SAVED CIX  
A37B  85F2  STA  CIX ; PUT BACK  
A37D  84AC  STY  TVSCIX ; SAVE NEW CIX  
        ;  
A37F  A5B3  LDA  VNTP+1 ; SEARCH VNT  
A381  A402  LDY  VNTP ; FOR THIS GUY  
A383  A200  LDX  #0  
A385  2062A4  JSR  SEARCH ;  
A388  B00A  "A394  BCS  :TVS0 ; BR NOT FOUND  
A38A  E4AC  CPX  TVSCIX ; FOUND RIGHT ONE  
A38C  F04D  "A3DB  BEQ  :TVSUC ; BR IF YES  
A38E  2098A4  JSR  SRCNXT ; GO SEARCH MORE  
A391  4888A3  JMP  :TVRS ; TEST THIS RESULT  
        ;  
A394  38  SEC ; SIGH:  
A395  A5AC  LDA  TVSCIX ; VAR LENGTH IS  
A397  E5F2  SBC  CIX ; NEW CIX-OLD CIX  
A399  85F2  STA  CIX ;  
A39B  A8  TAY ; GO EXPAND VNT  
A39C  A284  LDX  #$VNTD ; BY VAR LENGTH  
A39E  207FA8  JSR  EXPLow ; SET VARIABLE NUMBER  
A3A1  A5AF  LDA  STENUM ;  
A3A3  85D3  STA  TVNUM ;  
        ;  
A3A5  A4F2  LDY  CIX ; AND  
A3A7  88  DEY  
A3A8  A6AC  LDX  TVSCIX ; GET DISPL TO EQU+1  
A3AA  CA  DEX  
A3AB  B08005  :TVS1  LDA  LBUFF,X ; MOVE VAR TO  
A3AE  9197  STA  [SVESA],Y  
A3B0  CA  DEX  
A3B1  88  DEY  
A3B2  10F7  "A3AB  BPL  :TVS1 ;  
        ;  
A3B4  A4F2  LDY  CIX ;TURN ON MSB  
A3B6  88  DEY ; OF LAST CHAR  
A3B7  B197  LDA  [SVESA],Y ; IN VTVT ENTRY  
A3B9  09B0  ORA  #$80 ;  
A3BB  9197  STA  [SVESA],Y  
        ;  
A3BD  A008  LDY  #8 ; THEN EXPAND  
A3BF  A288  LDX  #$STMTAB ; VVT BY 8  
A3C1  207FA8  JSR  EXPLow  
A3C4  E6B1  INC  SVVVTE ; INC VVT EXP SIZE  
        ;  
A3C6  A002  LDY  #2 ; CLEAR VALUE  
A3CB  A900  LDA  #0 ; PART OF  
A3CA  99D200  :TVSIA  STA  TVTYPE,Y ; ENTRY  
A3CD  C8  INY  
A3CE  C088  CPY  #8  
A3D0  90F8  "A3CA  BCC  :TVS1A ; AND THEN  
A3D2  88  DEY  
A3D3  B9D200  :TVS2  LDA  TVTYPE,Y ; PUT IN VAR TABLE  
A3D6  9197  STA  [SVESA],Y ; ENTRY  
A3D8  88  DEY  
A3D9  10F8  "A3D3  BPL  :TVS2 ;
Source Code

A3DB 24D2 :TVSUC BIT TVTYPE ; WAS THERE A PAREN
A3DD 5002 *A3E1 BVC :TVNP ; BR IF NOT
A3DF 6AC DEC TVSCIX ; LET SYNTAX SEE PAREN

A3E1 A5AC :TVNP LDA TVSCIX ; GET NEW CIX
A3E3 85F2 STA CIX ; TO CIX

A3E5 A5AF LDA STENUM ; GET TABLE ENTRY NO
A3E7 3807 *A3F0 BMI :TVFULL ; BR IF > $7F
A3E9 9580 ORA #$80 ; MAKE IT > $7F
A3EB 28C8A2 JSR :SETCODE ; SET CODE TO OUTPUT BUFFER
A3EE 18 CLC ; SET SUCCESS CODE
A3EF 60 RTS ; RETURN

A3F0 4C38B9 :TVFULL JMP ERRVSF ; GO TO ERROR RTN

TSTALPH

; TEST CIX FOR ALPHA
A3F3 14F2 LDA CIX
A3F5 14F3 LDA [INBUFF],Y
A3F7 TSTALPH
A3F7 C941 CMP #A
A3F9 9003 *A3FE BCC :TFAIL
A3FB C95B CMP #$3B
A3FD 60 RTS

A3FE 38 :TFAIL SEC
A3FF 60 RTS

TNCON

; EXTERNAL SUBROUTINE TO CHECK FOR NUMBER
A400 20A1DB JSR SKBLANK
A403 5AF2 LDA CIX
A405 85AC STA TVSCIX
A407 2800D8 JSR CVAPP ; GO TEST AND CONV
A40A 9005 *A411 BCC :TNC1 ; BR IF NUMBER
A40C A5AC LDA TVSCIX
A40E 85F2 STA CIX ; RETURN FAIL
A410 60 RTS

A411 A90E :TNC1 LDA #$9E ; SET NUMERIC CONST
A413 20C8A2 JSR :SETCODE

A416 4A94 LDX COX
A418 A200 LDX #0
A41A B5D4 :TNC2 LDA FR0,X ; MOVE CONST TO STMT
A41C 9100 STA [OUTBUFF],Y
A41E C8 INY
A41F 88 INX
A420 E806 CPX #6
A422 9E06 *A41A BCC :TNC2
A424 4944 STY COX
A426 18 CLC
A427 60 RTS

TSCON

; EXT RTN TO CHECK FOR STR CONST
A428 20A1DB JSR SKBLANK ; GET INDEX
A42B A4F2 LDA CIX ; GET CHAR
A42D B1F3 LDA [INBUFF],Y ; IS IT DQUOTE
A42F C922 CMP #$22 ; BR IF DQ
A431 F802 *A435 BEQ :TSC1 ; SET FAIL
A433 38 SEC ; RETURN
A434 60 RTS ;
Source Code

```
A435 A90F :TSC1 LDA #$8F ; SET SCON CODE
A437 2BC8A2 JSR :SETCODE
A43A A594 LDA COX ; SET COX
A43C B5AB STA TSCOX ; SAVE FOR LENGTH
A43E 2BC8A2 JSR :SETCODE ; SET DUMMY FOR NOW

A441 B6P2 :TSC2 INC CIX ; NEXT INPUT CHAR
A443 A4F2 LDY CIX
A445 B1F3 LDA [INBUFF],Y
A447 C99B CMP #$CR ; IS IT CR
A449 F0C A457 BEQ :TSC4 ; BR IF CR
A44B C922 CMP #$82 ; IS IT DQ
A44D F06 A455 BEQ :TSC3 ; BR IF DQ
A44F 2BC8A2 JSR :SETCODE ; OUTPUT IT
A452 4C41A4 JMP :TSC2 ; NEXT

A455 B6P2 :TSC3 INC CIX ; INC CIX OVER DQ
A457 18 :TSC4 CLC
A45B A594 LDA COX ; LENGTH IS COX MINUS
A45A E5AB SRC TSCOX ; LENGTH BYTE COX
A45C A4AB LDY TSCOX
A45E 9180 STA [OUTBUFF],Y ; SET LENGTH

A460 18 CLC ; SET SUCCESS
A461 60 RTS ; DONE

Search a Table

TABLE FORMAT:
GARBAGE TO SKIP [N]
ASCII CHAR [N]
WITH LEAST SIGNIFICANT BYTE HAVING
MOST SIGNIFICANT BIT ON
LAST TABLE ENTRY MUST HAVE FIRST ASCII
CHAR = 0
ENTRY PARGS:
X = SKIP LENGTH
A,Y = TABLE ADR [HIGH LOW]
ARGUMENT = INBUFF + CIX
EXIT PARGS:
CARRY = CLEAR IF FOUND
X = FOUND ARGUMENT END CIX+1
SRCADR = TABLE ENTRY ADR
STENUM = TABLE ENTRY NUMBER

A462 SEARCH
A462 86AA STX SRCSKIP ; SAVE SKIP FACTOR
A464 A2FF LDSX #$FF ; SET ENTRY NUMBER
A466 86AF STX STENUM ; TO ZERO

A468 B596 :SRC1 STA SRCADR+1 ; SET SEARCH ADR
A46A B495 STY SRCADR
A46C E6AF INC STENUM
A46E A6F2 LDX CIX
A470 A4AA LDY SRCSKIP
A472 B195 LDA [SRCADR],Y
A474 F027 A49D BEQ :SRC5 ; BR IF EOT
A476 A900 LDA #$0 ; SET STATUS = EQ
A478 8B PHP
A479 BB805 :SRC2 LDA LBUFF,X ; GET INPUT CHAR
A47C 29F AND #$7F ; TURN OFF MSB
A47E C92E CMP #'.'; IF WILD CARD
A480 F01D A49F BEQ :SRC5 ; THEN BR
A482 :SRC2A
A482 5195 EOR [SRCADR],Y ; EX-OR WITH TABLE CHAR
A484 ASLA A ; SHIFT MSB TO CARRY
A484 +8A ASL A
A485 F002 A489 BEQ :SRC3 ; BR IF [ARG=TAB] CHAR
```

---

158
Source Code

A487 68 PLA ; POP STATUS
A488 08 PHP ; PUSH NE STATUS

A489 CB :SRC3 INY ; INC TABLE INDEX
A48A E8 INX
A48B 90EC A479 BCC :SRC2 ; IF TABLE MSB OFF, CONTINUE
A48D 28 PLF
A48E F00B A49B BEQ :SRCFND ; BR IF NO MIS MATCH

A490 SRCNXT
A491 18 CLC
A492 90 TYA
A494 A8 TAY
A495 A596 LDA SRCADR+1
A497 6900 ADC #0
A499 D0CD A468 BNE :SRC1 ; BR ALLWAYS

A49B 18 :SRCFND CLC ; INDICATE FOUND
A49C 60 RTS
A49D 38 :SRCNF SEC ; INDICATE NOT FOUND
A49E 60 RTS

A49F A902 :SRC5 LDA #2 ; IF NOT
A4A1 C5AA CMP SRCKP ; STMT NAME TABLE
A4A3 D0DD A482 BNE :SRC2A ; THEN IGNORE
A4A5 B195 :SRC6 LDA [SRCADR],Y ; TEST MSB OF TABLE
A4A7 3003 A4AC BMI :SRC7 ; IF ON DONE
A4A9 C8 INY
A4AA D0F9 A4A5 BNE :SRC6 ; LOOK AT NEXT CHAR
A4AC 38 :SRC7 SEC ; INDICATE MSB ON
A4AD B0DA A489 BCS :SRC3 ; AND RE-ENTER CODE

Statement Name Table

; SNTAB- STATEMENT NAME TABLE
; EACH ENTRY HAS SYNTAX TABLE ADR PTR
; FOLLOWED BY STMT NAME

A4AF SNTAB

A4AF C7A7 DW :SREM-1
A4B1 5245CD DC 'REM'
A4B4 CAA7 DW :SDATA-1
A4B6 444154C1 DC 'DATA'
A4BA P3A6 DW :SINPUT-1
A4BC 494E5055D4 DC 'INPUT'
A4C1 BCA6 DW :SCOLOR-1
A4C3 434F4C4FD2 DC 'COLOR'
A4C8 32A7 DW :SLIST-1
A4CA 4C4953D4 DC 'LIST'
A4CE 23A7 DW :SENDER-1
A4D0 454E5445D2 DC 'ENTER'
A4D5 BFA6 DW :SLET-1
A4D7 4C45D4 DC 'LET'
A4DA 93A7 DW :SIP-1
A4DC 49C6 DC 'IF'
A4DE D1A6 DW :SFOR-1
A4E0 464FD2 DC 'FOR'
A4E3 E9A6 DW :SNEXT-1

159
Source Code

A4E5  4E4550D4  DC  'NEXT'
A4E9  BCA6   DW    :SGOTO-1
A4EB  474F54CF  DC  'GOTO'
A4EF  BCA6   DW    :SGOTO-1
A4F1  474F2854CF  DC  'GO TO'
A4F6  BCA6   DW    :SGOSUB-1
A4F8  474F5355C2  DC  'GOSUB'
A4FD  BCA6   DW    :STRAP-1
A4FF  545241D0  DC  'TRAP'

A503  BDA6   DW    :SBYE-1
A505  4259C5  DC  'BYE'
A508  BDA6   DW    :SCONT-1
A50A  3434F4ED4  DC  'CONT'
A50E  58A7   DW    :SCOM-1
A510  434FCD  DC  'COM'

A513  20A7   DW    :SCLOSE-1
A515  434C4F53C5  DC  'CLOSE'
A51A  BDA6   DW    :SCLR-1
A51C  343CD2  DC  'CLR'
A51F  BDA6   DW    :SDIM1-1
A521  4445C7  DC  'DEG'

A524  5FA7   DW    :SDIM-1
A526  4449CD  DC  'DIM'
A529  BDA6   DW    :SEND-1
A52B  454EC4  DC  'END'

A52E  BDA6   DW    :SNOW-1
A530  4E45D7  DC  'NEW'

A533  19A7   DW    :SOPEN-1
A535  4F5045CE  DC  'OPEN'
A539  23A7   DW    :SLOAD-1
A53B  4C4F41C4  DC  'LOAD'
A53F  23A7   DW    :SSAVE-1
A541  534156C5  DC  'SAVE'
A545  48A7   DW    :SSTATUS-1
A547  5354415455  DC  'STATUS'
A54D  49A7   DW    :SNOTE-1
A54F  4E4F54C5  DC  'NOTE'
A553  49A7   DW    :SPCONST-1
A555  504494ED4  DC  'POINT'
A55A  17A7   DW    :SXIO-1
A55C  5849CF  DC  'XIO'

A55F  62A7   DW    :SON-1
A561  4FCE   DC  'ON'

A563  5CA7   DW    :SPOKE-1
A565  584F4BC5  DC  'POKE'
A569  FBA6   DW    :SPRINT-1
A56B  5852494ED4  DC  'PRINT'
A570  BDA6   DW    :SRAD-1
A572  5241C4  DC  'RAD'
A575  P4A6   DW    :SREAD-1
Source Code

A577  524541C4  DC 'READ'
A57B  EEA6  DW :SREST-1
A57D  524553544F  DC 'RESTORE'
A584  BDA6  DW :SRET-1
A586  5245545552  DC 'RETURN'
A58C  26A7  DW :SRUN-1
A58E  5255CE  DC 'RUN'
A591  BDA6  DW :SSTOP-1
A593  53544FD0  DC 'STOP'
A597  BDA6  DW :SPOP-1
A599  584FD0  DC 'POP'
A59C  FBA6  DW :SPRINT-1
A59E  BF  DC '?'
A59F  E7A6  DW :SGET-1
A5A1  4745D4  DC 'GET'
A5A4  B9A6  DW :SPUT-1
A5A6  5855D4  DC 'PUT'
A5A9  BCA6  DW :SGR-1
A5AB  4752415048  DC 'GRAPHICS'
A5B1  5CA7  DW :SPL0T-1
A5B5  584C4FD4  DC 'PLOT'
A5B9  5CA7  DW :SPOS-1
A5BB  504F534954  DC 'POSITION'
A5C3  BDA6  DW :SDOS-1
A5C5  444FD3  DC 'DOS'
A5C8  5CA7  DW :SDRAWTO-1
A5CA  4452415754  DC 'DRAWTO'
A5D0  5AA7  DW :SSETCOLOR-1
A5D2  534554434F  DC 'SETCOLOR'
A5DA  E1A6  DW :SLOCATE-1
A5DC  4C4F34154F  DC 'LOCATE'
A5E2  58A7  DW :SSOUND-1
A5E4  534F554EC4  DC 'SOUND'
A5E9  FFA6  DW :SLPRINT-1
A5EF  4C5052494E  DC 'LPRINT'
A5F1  BDA6  DW :SCSAVE-1
A5F3  43534156C5  DC 'CSAVE'
A5FB  BDA6  DW :SLOAD-1
A5FA  434C4F41C4  DC 'CLOAD'
A5FF  FFA6  DW :SILET-1
A601  00  DB 0
A602  8000  DB $80,00
A604  2A4552524F  DB '*ERROR-
A60C  A0  DB $A0

161
### Syntax Tables

#### Syntax Table OP Codes

<table>
<thead>
<tr>
<th>OP Code</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 EQU $00</td>
<td>ABSOLUTE NON TERMINAL VECTOR FOLLOWED BY 2 BYTE ADR -1</td>
</tr>
<tr>
<td>0001 EQU $01</td>
<td>EXTERNAL SUBROUTINE CALL FOLLOWED BY 2 BYTE ADR -1</td>
</tr>
<tr>
<td>0002 EQU $02</td>
<td>ALTERNATIVE, BNF OR ( )</td>
</tr>
<tr>
<td>0003 EQU $03</td>
<td>RETURN, ( )</td>
</tr>
<tr>
<td>0004 EQU $04</td>
<td>ACCEPT TO THIS POINT ( &amp; )</td>
</tr>
<tr>
<td>0005 EQU $05</td>
<td>SPECIAL NTV FOR EXP ( &lt;EXP&gt; )</td>
</tr>
<tr>
<td>0006 EQU $06</td>
<td>CHANGE LAST OUTPUT TOKEN</td>
</tr>
</tbody>
</table>

### Code Examples

```
<EXP> = ( <EXP> ) <NOP> | <UNARY> <EXP> | <NV> <NOP> #
```

### Unary Operators

<table>
<thead>
<tr>
<th>Code</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>A60D</td>
<td>EXP SYN CLPRN</td>
</tr>
<tr>
<td>A60E</td>
<td>SYM JS, EXP</td>
</tr>
<tr>
<td>A60F</td>
<td>SYM CRPRN</td>
</tr>
<tr>
<td>A610</td>
<td>SYN JS, NOP</td>
</tr>
<tr>
<td>A611</td>
<td>SYN OR</td>
</tr>
<tr>
<td>A612</td>
<td>SYN JS, UNARY</td>
</tr>
<tr>
<td>A613</td>
<td>SYN JS, EXP</td>
</tr>
<tr>
<td>A614</td>
<td>SYN OR</td>
</tr>
<tr>
<td>A615</td>
<td>SYN JS, NV</td>
</tr>
<tr>
<td>A616</td>
<td>SYN JS, NOP</td>
</tr>
<tr>
<td>A617</td>
<td>SYN RTN</td>
</tr>
</tbody>
</table>

### Not Operators

<table>
<thead>
<tr>
<th>Code</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>A618</td>
<td>UNARY SYN CPLUS</td>
</tr>
<tr>
<td>A619</td>
<td>SYN CHNG, CPLUS</td>
</tr>
<tr>
<td>A620</td>
<td>SYN CHNG</td>
</tr>
<tr>
<td>A621</td>
<td>SYN CNOT</td>
</tr>
<tr>
<td>A622</td>
<td>SYN JS, :NFUN, :OR</td>
</tr>
<tr>
<td>A623</td>
<td>SYN :OR</td>
</tr>
<tr>
<td>A624</td>
<td>SYN JS, NVAR, :OR</td>
</tr>
<tr>
<td>A625</td>
<td>SYN :OR</td>
</tr>
<tr>
<td>A626</td>
<td>SYN ESRT, AD, :TNCON-1, :OR</td>
</tr>
</tbody>
</table>

### NV Operators

<table>
<thead>
<tr>
<th>Code</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>A622</td>
<td>NV SYN JS, :NFUN, :OR</td>
</tr>
<tr>
<td>A623</td>
<td>SYN :OR</td>
</tr>
<tr>
<td>A624</td>
<td>SYN JS, NVAR, :OR</td>
</tr>
<tr>
<td>A625</td>
<td>SYN :OR</td>
</tr>
<tr>
<td>A626</td>
<td>SYN :ESRT, AD, :TNCON-1, :OR</td>
</tr>
</tbody>
</table>
Source Code

A627 +FFA3  DW  (:TNCON-1)
A629 +02  DB  :OR
A62A  SYN  :ANTV, AD, :STCOMP-1
A62A +00  DB  :ANTV
A62B +7DA6  DW  (:STCOMP-1)
A62D  SYN  :RTN
A62D +03  DB  :RTN

< NOP > = < OP > < EXP > | &#

A62E  :NOP  SYN  :JS, :OP
A62E +C4  DB  $B0+(((:OP-*)&$7F) XOR $40 )
A62F  SYN  :JS, :EXP
A62F +9E  DB  $B0+(((:EXP-*)&$7F) XOR $40 )
A630  SYN  :OR
A630 +02  DB  :OR
A631  DB  :RTN
A631 +03  DB  :RTN

< OP > = ** | * | / | = | S = | < > | < > = | AND | OR#

A632  :OP  SYN  :CEXP, :OR
A632 +23  DB  :CEXP
A633 +02  DB  :OR
A634  DB  :OR
A634 +25  DB  :CPLUS
A635 +02  DB  :OR
A636  DB  :CMINUS
A636 +26  DB  :CMINUS
A637 +02  DB  :OR
A638  DB  :OR
A638 +24  DB  :CMUL
A639 +02  DB  :OR
A63A  DB  :OR
A63A +27  DB  :CDIV
A63B +02  DB  :OR
A63C  DB  :OR
A63C +1D  DB  :CLE
A63D +02  DB  :OR
A63E  DB  :OR
A63F +1F  DB  :CGE
A63F +02  DB  :OR
A640  DB  :OR
A640 +1E  DB  :CNE
A641 +02  DB  :OR
A642  DB  :OR
A642 +20  DB  :CLT
A643 +02  DB  :OR
A644  DB  :CLT
A644 +21  DB  :CGT
A645 +02  DB  :OR
A646  DB  :OR
A646 +22  DB  :CEQ
A647 +02  DB  :OR
A648  DB  :OR
A648 +2A  DB  :CAND
A649 +02  DB  :OR
A64A  DB  :OR
A64A +29  DB  :COR
A64B  DB  :RTN
A64B +03  DB  :RTN

< NVAR > = < TNVAR > < NMAT > #
Source Code

\[
<N\text{MAT}>
= ( \langle EXP \rangle \langle NMAT2 \rangle ) \ | \ &\# \\
A651~+2B & DB CLPRN, :CHNG, CALPRN \\
A652~+0F' & DB :CHNG \\
A653~+3B & DB CALPRN \\
A654 & SYN :VEXP \\
A655~+0E & DB :VEXP \\
A655~+C4 & DB $80+((::NMAT2-*)&$7F) XOR $40$ \\
A656 & SYN CRPRN \\
A657 & SYN :OR \\
A657~+02 & DB :OR \\
A658 & SYN :RTN \\
A658~+03 & DB :RTN \\
\]

\[
<N\text{MAT2}>
= ,\langle EXP \rangle \ | \ &\# \\
A659~+12 & DB CCOM \\
A65A~+0F & DB :CHNG \\
A65B~+3C & DB CACOM \\
A65C ~ SYN :VEXP \\
A65C~+0E & DB :VEXP \\
A65D & SYN :OR \\
A65D~+02 & DB :OR \\
A65E & SYN :RTN \\
A65E~+03 & DB :RTN \\
\]

\[
<N\text{FUN}>
= \langle NFNP \rangle \langle NFP \rangle \ | \ <NFSP \rangle \langle SFP \rangle \ | \ <NFUSR>\# \\
A65F ~ SYN :CFNP \\
A660 & DB CNFNP \\
A660 & SYN :JS , :NFPP \\
A66B~+D2 & DB $80+((::NFP--*)&$7F) XOR $40$ \\
A661 & SYN :OR \\
A661~+02 & DB :OR \\
A662 & SYN :ANTV,AD,:NFSP-1 \\
A663~+00 & DB :ANTV \\
A663~+CDN7 & DW {::NFSP-1} \\
A665~+03 & DB $80+((::SFP-*)&$7F) XOR $40$ \\
A666 & SYN :OR \\
A666~+02 & DB :OR \\
A667 & SYN :JS , :NFUSR \\
A667~+C2 & DB $80+((::NFUSR--*)&$7F) XOR $40$ \\
A668 & SYN :RTN \\
A668~+03 & DB :RTN \\
\]

\[
<N\text{USR}>
= USR ( <PUSR> ) \# \\
A669 & DB CUSR \\
A669~+3F & DB CCLPRN, :CHNG, CFLPRN \\
A66A~+2B & DB :CHNG \\
A66B~+0F & DB CFLPRN \\
A66C~+3A & DB :ANTV \\
A66D~+00 & DB :ANTV \\
A66E~+D9A7 & DW {::PUSR-1} \\
A670 & SYN CRPRN \\
A670~+2C & DB CRPRN \\
A671 & SYN :RTN \\
A671~+03 & DB :RTN \\
\]

\[
<NFP>
= ( \langle EXP \rangle \# \\
A672~+2B & DB CLPRN \\
A672~+0F & DB :CHNG \\
A674~+3A & DB CFLPRN \\
A675 & SYN :VEXP \\
\]
A675 +9E  DB : VEXP
A676 SYN CRPRN
A676 +2C  DB CRPRN
A677 SYN : RTN
A677 +03  DB : RTN

< SFP > = < STR > ) #
A678 : SFP SYN CLPRN, : CHNG, CPLPRN
A678 +2B  DB CLPRN
A679 +0F  DB : CHNG
A67A +3A  DB CPLPRN
A67B SYN JS, : STR
A67B +C7  DB $80+(((:STR-*)&$7F) XOR $40)
A67C SYN CRPRN
A67C +2C  DB CRPRN
A67D SYN : RTN
A67D +03  DB : RTN

< STCOMP > = < STR > < SOP > < STR > #
A67E : STCOMP SYN JS, : STR
A67E +C4  DB $80+(((:STR-*)&$7F) XOR $40)
A67F SYN JS, : SOP
A67F +E3  DB $80+(((:SOP-*)&$7F) XOR $40)
A680 SYN JS, : STR
A680 +C2  DB $80+(((:STR-*)&$7F) XOR $40)
A681 SYN : RTN
A681 +03  DB : RTN

< STR > = < SFUN > | < SVAR > | < SCON > #
A682 : STR SYN JS, : SFUN
A682 +C8  DB $80+(((:SFUN-*)&$7F) XOR $40)
A683 SYN : OR
A683 +02  DB : OR
A684 SYN JS, : SVAR
A684 +C8  DB $80+(((:SVAR-*)&$7F) XOR $40)
A685 SYN : OR
A685 +02  DB : OR
A686 SYN : ESRT, AD, : TSCON-1
A686 +01  DB : ESRT
A687 +27A4 DW (: TSCON-1)
A688 SYN : RTN
A689 +03  DB : RTN

< SFUN > = SFNP < NFP > #
A68A : SFUN SYN : ANTV, AD, : SFNP-1
A68A +00  DB : ANTV
A68B +D5A7 DW (: SFNP-1)
A68D SYN JS, : NFP
A68D +A5  DB $80+(((:NFP-*)&$7F) XOR $40)
A68E SYN : RTN
A68E +03  DB : RTN

< SVAR > = < TSVAR > < SMAT > #
A68F : SVAR SYN : ESRT, AD, : TSVAR-1
A68F +01  DB : ESRT
A690 +2DA3 DW (: TSVAR-1)
A692 SYN JS, : SMAT
A692 +C2  DB $80+(((:SMAT-*)&$7F) XOR $40)
A693 SYN : RTN
A693 +03  DB : RTN

< SMAT > = ( < EXP > < SMAT2 > ) & #
A694 : SMAT SYN CLPRN, : CHNG, CSLPRN
A694 +2B  DB CLPRN
A695 +0F  DB : CHNG
A696 +37  DB CSLPRN
Source Code

A697 SYN :VEXP
A697 +OE DB :VEXP
A698 SYN JS, :SMAT2
A698 +C4 DB $80+((:(SMAT2-*)&$7F) XOR $40 )
A699 SYN CRPRN
A699 +2C DB CRPRN
A69A SYN :OR
A69A +02 DB :OR
A69B SYN :RTN
A69B +03 DB :RTN

<SMAT2> = , <EXP> | &#

A69C SYN :SMAT2 SYN CCOM, :CHNG, CACOM
A69C SMAT2 DB CCOM
A69D +0F DB :CHNG
A69F +2C DB :VEXP
A69F +OE DB :VEXP
A6AF SYN :OR
A6AF +02 DB :OR
A6AH SYN :RTN
A6AH +03 DB :RTN

<SOP> = < > <#>

A6A2 SYN :SOP
A6A2 SPDB CLE, :CHNG, CSLE, :OR
A6A2 +1D DB CLE
A6A3 +OE DB :CHNG
A6A4 +2F DB CSLE
A6A5 +02 DB :OR
A6A6 SYN CNE, :CHNG, CSNE, :OR
A6A6 +1E DB CNE
A6A6 +0F DB :CHNG
A6A8 +3B DB CSNE
A6A9 +02 DB :OR
A6AA SYN CGE, :CHNG, CSGE, :OR
A6AA +1F DB CGE
A6AA +0F DB :CHNG
A6AC +31 DB CGE
A6AD +02 DB :OR
A6AE SYN CLT, :CHNG, CSLT, :OR
A6AE +20 DB CLT
A6AF +0F DB :CHNG
A6B0 +32 DB CSLT
A6B1 +02 DB :OR
A6B2 SYN CGT, :CHNG, CSGT, :OR
A6B2 +21 DB CGT
A6B3 +0F DB :CHNG
A6B4 +33 DB CGT
A6B5 +02 DB :OR
A6B6 SYN CEQ, :CHNG, CSEQ
A6B6 +22 DB CEQ
A6B7 +0F DB :CHNG
A6B8 +34 DB CEQ
A6B9 SYN :RTN
A6B9 +03 DB :RTN

<PUT> = <D1>, <EXP> <EOS> #

A6BA SYN CPND, :VEXP
A6BA +1C DB CPND
A6BB +OE DB :VEXP
A6BC SYN CCOM
A6BC +12 DB CCOM
Source Code

< > = <EXP> <EOS>#

A6BD : STRAP
A6BD : SGOTO
A6BD : SGOSUB
A6BD : SGR
A6BD : SCOLOR
A6BD : XEOS SYN : VEXP
A6BD +0E DB : VEXP

< > = <EOS>#

A6BE : SCSAVE
A6BE : SCLOAD
A6BE : SDS
A6BE : SCLR
A6BE : SRET
A6BE : SEND
A6BE : SSTOP
A6BE : SPOP
A6BE : SNEW
A6BE : SBYE
A6BE : SCONT
A6BE : SDEG
A6BE : SRA
A6BE SYN JS, : EOS
A6BE +PA DB $80+(((:EOS-*)&$7F) XOR $40 )
A6BF SYN : RTN
A6BF +03 DB : RTN

< LET> = < NVAR> = <EXP> < EOS> | < SVAR> = < STR> <EOS> #

A6C0 : SLET
A6C0 : SILET
A6C0 SYN : ANTV, AD, : NVAR-1
A6C0 +00 DB : ANTV
A6C1 +4BA6 DW (: NVAR-1)
A6C3 SYN : CEQ, : ChNG, CAASN
A6C3 +22 DB : CEQ
A6C4 +0F DB : CHNG
A6C5 +2D DB : CAASN
A6C6 SYN : VEXP
A6C6 +0E DB : VEXP
A6C7 SYN : JS, : EOS
A6C7 +P1 DB $80+(((:EOS-* )&$7F) XOR $40 )
A6C8 SYN : OR
A6C8 +02 DB : OR

A6C9 SYN : JS, : SVAR
A6C9 +06 DB $80+(((:SVAR-*)&$7F) XOR $40 )
A6CA SYN : CEQ, : ChNG, CAASN
A6CA +22 DB : CEQ
A6CB +0F DB : CHNG
A6CC +2E DB : CAASN
A6CD SYN : ANTV, AD, : STR-1
A6CD +00 DB : ANTV
A6CE +81A6 DW (: STR-1)
A6D0 SYN JS, : EOS
A6D0 +08 DB $80+(((:EOS-* )&$7F) XOR $40 )
A6D1 SYN : RTN
A6D1 +03 DB : RTN

< FOR> = < TNVAR> = <EXP> TO <EXP> < FSTEP> <EOS> #

A6D2 : SFOR SYN : ESRT, AD, : TNVAR-1
A6D2 +01 DB : ESRT
A6D3 +29A3 DW (: TNVAR-1)
A6D5 SYN : CEQ, : ChNG, CAASN
A6D5 +22 DB : CEQ
A6D6 +0F DB : CHNG
A6D7 +2D DB : CAASN
A6D9 SYN :VEXP
A6DB +0E DB :VEXP
A6D9 SYN CTO
A6D9 +19 DB CTO
A6DA SYN :VEXP
A6DA +0E DB :VEXP
A6DB SYN JS, FSTEP
A6DB +C3 DB $80+(((:FSTEP-*)&$7F) XOR $40)
A6DC SYN JS, EOS
A6DC +DC DB $80+(((:EOS-*)&$7F) XOR $40)
A6DD SYN :RTN
A6DD +03 DB :RTN

<FSTEP> = STEP <EXP> | &
A6DE :FSTEP
A6DE SYN CSTEP
A6DF +1A DB CSTEP
A6DF +0E DB :VEXP
A6D0 +02 DB :OR
A6E1 SYN :RTN
A6E1 +03 DB :RTN

<LOCATE> = <EXP>, <EXP>, <TNVAR> <EOL>
A6E2 :SLOCATE
A6E2 SYN :VEXP
A6E2 +0E DB :VEXP
A6E3 SYN CCOM
A6E3 +12 DB CCOM
A6E4 SYN :VEXP
A6E4 +0E DB :VEXP
A6E5 SYN CCOM
A6E5 +12 DB CCOM
A6E6 SYN JS, SNEXT
A6E6 +C4 DB $80+(((:SNEXT-*)&$7F) XOR $40)
A6E7 SYN :RTN
A6E7 +03 DB :RTN

<GET> = <D1>, <TNVAR>
A6E8 :SGET
A6E8 SYN JS, D1
A6E9 +DD DB $80+(((:D1-*)&$7F) XOR $40)
A6E9 SYN CCOM
A6E9 +01 DB CCOM

<NEXT> = <TNVAR> <EOS>
A6EA +01 DB ESRT, AD, :TNVAR-1
A6EB +29A3 DW (:TNVAR-1)
A6ED SYN JS, EOS
A6ED +CB DB $80+(((:EOS-*)&$7F) XOR $40)
A6EE SYN :RTN
A6EE +03 DB :RTN

<RESTORE> = <EXP> <EOS> | <EOS>
A6EF :SREST SYN :VEXP
A6EF +0E DB :VEXP
A6F0 SYN JS, EOS
A6F0 +C8 DB $80+(((:EOS-*)&$7F) XOR $40)
A6F1 SYN :OR
A6F1 +02 DB :OR
A6F2 SYN JS, EOS
A6F2 +C6 DB $80+(((:EOS-*)&$7F) XOR $40)
A6F3 SYN :RTN
A6F3 +03 DB :RTN
<INPUT> = <OPD> <READ>#
A6F4 :INPUT  SYN JS, ;OPD
A6F4 +FB DB $80+((:(OPD-*)&$7F) XOR $40 )

<READ> = <NSVRL> <EOS>#
A6F5 :READ  SYN JS, ;NSVRL
A6F5 +DB DB $80+(((NSVRL-*)&$7F) XOR $40 )
A6F6 SYN JS, ;EOS
A6F6 +C2 DB $80+((:(EOS-*)&$7F) XOR $40 )
A6F7 SYN :RTN
A6F7 +03 DB :RTN

EOS =: | CR#
A6F8 :EOS SYN CEOS
A6F8 +14 DB CEOS
A6F9 SYN :OR
A6F9 +02 DB :OR
A6FA SYN CCR
A6FA +16 DB CCR
A6FB SYN :RTN
A6FB +03 DB :RTN

<PRINT> = <D1> <EOS> | <D1> <PR1> <EOS>
A6FC :PRINT
A6FC SYN JS, ;D1
A6FC +C9 DB $80+(((D1-*)&$7F) XOR $40 )
A6FD SYN JS, ;EOS
A6FD +BB DB $80+((:(EOS-*)&$7F) XOR $40 )
A6FE SYN :OR
A6FE +02 DB :OR
A6FF SYN JS, ;OPD
A6FF +ED DB $80+(((:(OPD-*)&$7F) XOR $40 )
A700 :SLPRINT
A700 SYN :ANTV, AD, ;PR1-1
A700 +00 DB :ANTV
A701 +9FA7 DW (:PR1-1)
A703 SYN JS, ;EOS
A703 +B5 DB $80+(((EOS-*)&$7F) XOR $40 )
A704 SYN :RTN
A704 +03 DB :RTN

<D1> = <CPND> <EXP>#
A705 :D1 SYN CPND
A705 +1C DB CPND
A706 SYN :VEXP
A706 +0E DB :VEXP
A707 SYN :RTN
A707 +03 DB :RTN

<NSVAR> = <NVAR> | <SVAR>#
A708 :NSVAR SYN ESRT, AD, ;TNVAR-1
A708 +01 DB :ESRT
A709 +29A3 DW (:TNVAR-1)
A70B SYN :OR
A70B +02 DB :OR
A70C SYN ESRT, AD, ;TSVAR-1
A70C +01 DB :ESRT
A70D +2DA3 DW (:TSVAR-1)
A70F SYN :RTN
A70F +03 DB :RTN

<NSVRL> = <NSVAR> <NSV2> | &#
A710 :NSVRL SYN JS, ;NSVAR
A710 +B8 DB $80+((:(NSV2-*)&$7F) XOR $40 )
A711 SYN JS, ;NSV2

169
Source Code

A711 +C3    DB $80+((::NSV2-*)&$7F) XOR $40
A712 SYR   :OR, :RTN
A712 +03    DB :OR
A713 +03    DB :RTN

<NSV2> = <NSVRL> | &#
A714 +12    DB SYN CCOM
A715 SYR   JS, :NSVRL
A715 +BB    DB $80+((::NSVRL-*)&$7F) XOR $40
A716 SYR   :OR, :RTN
A716 +02    DB :OR
A717 +03    DB :RTN

<XIO> = <AEXP>, <D2S>, <FS>, <AEXP>, <EOS>#
A718 :SXIO  SYR   :VEXP
A718 +0E    DB :VEXP
A719 SYR   CCOM
A719 +12    DB CCOM

<OPEN> = <D1>, <EXP>, <EXP>, <FS>, <EOS>#
A71A :SOPEN SYR   JS, :D1
A71A +AB    DB $80+((::D1-*)&$7F) XOR $40
A718 SYR   CCOM
A718 +12    DB CCOM
A71C SYR   JS, :TEXP
A71C +F9    DB $80+((::TEXP-*)&$7F) XOR $40
A71D SYR   CCOM
A71D +12    DB CCOM
A71E SYR   JS, :FS
A71E +F3    DB $80+((::FS-*)&$7F) XOR $40
A71F SYR   JS, :EOS
A71F +99    DB $80+((::EOS-*)&$7F) XOR $40
A720 SYR   :RTN
A720 +03    DB :RTN

<CLOSE> = <D1> <EOS>#
A721 :SCLOSE SYR   JS, :D1
A721 +A4    DB $80+((::D1-*)&$7F) XOR $40
A722 SYR   JS, :EOS
A722 +96    DB $80+((::EOS-*)&$7F) XOR $40
A723 SYR   :RTN
A723 +03    DB :RTN

< > = <FS> <EOS>#
A724 :SENTER SYR   JS, :FS
A724 :SLOAD SYR   JS, :SAVE
A724 +0D    DB SYN JS, :FS
A724 +ED    DB $80+((::FS-*)&$7F) XOR $40
A725 SYR   JS, :EOS
A725 +93    DB $80+((::EOS-*)&$7F) XOR $40
A726 SYR   :RTN
A726 +03    DB :RTN

<RUN> = <FS> <EOS2> | <EOS2>#
A727 :SRUN SYR   JS, :FS
A727 +EA    DB $80+((::FS-*)&$7F) XOR $40
A728 SYR   JS, :EOS
A728 +90    DB $80+((::EOS-*)&$7F) XOR $40
A729 SYR   :OR
A729 +02    DB :OR
Source Code

A72A SYN JS, :EOS
A72A +0E DB $80+((:EOS-*)&$7F) XOR $40
A72B SYN :RTN
A72B +03 DB :RTN

< OPD > = < D1 >, | #
A72C :OPD
A72C SYN JS, :D1
A72C +99 DB $80+((:D1-*)&$7F) XOR $40
A72D :OPDX SYN CCOM
A72D +12 DB CCOM
A72E SYN :OR
A72E +02 DB :OR
A72F SYN JS, :D1
A72F +96 DB $80+((:D1-*)&$7F) XOR $40
A730 SYN CSC
A730 +15 DB CSC
A731 SYN :OR
A731 +02 DB :OR
A732 SYN :RTN
A732 +03 DB :RTN

< LIST > = < FS >; < L2 > | < L2 >#
A733 :SLIST
A733 SYN JS, :FS
A734 SYN JS, :EOS
A734 +04 DB $80+((:EOS-*)&$7F) XOR $40
A735 SYN :OR
A735 +02 DB :OR
A736 SYN JS, :FS
A736 +0B DB $80+((:FS-*)&$7F) XOR $40
A737 SYN CCOM
A737 +12 DB CCOM
A738 SYN JS, :LIS
A738 +0C4 DB $80+((:LIS-*)&$7F) XOR $40
A739 SYN :OR
A739 +02 DB :OR
A73A SYN JS, :LIS
A73A +02 DB $80+((:LIS-*)&$7F) XOR $40
A73B SYN :RTN
A73B +03 DB :RTN

< LIS > = < L1 > < EOS2 >#
A73C :LIS
A73C SYN :ANTV, AD, :L1-1
A73D +00 DB :ANTV
A73D +0FA7 DW (:L1-1)
A73F SYN JS, :EOS2
A73F +P4 DB $80+((:EOS2-*)&$7F) XOR $40
A740 SYN :RTN
A740 +03 DB :RTN

< STATUS > = < STAT > < EOS2 >#
A741 :SSTATUS
A741 SYN JS, :STAT
A741 +0C3 DB $80+((:STAT-*)&$7F) XOR $40
A742 SYN JS, :EOS2
A742 +0F1 DB $80+((:EOS2-*)&$7F) XOR $40
A743 SYN :RTN
A743 +03 DB :RTN

< STAT > = < D1 >, < NVAR >#
A744 :STAT
A744 SYN JS, :D1
A744 +81 DB $80+((:D1-*)&$7F) XOR $40
Source Code

A745 SYN CCOM
A745 +12 DB CCOM
A746 SYN :ANTV, AD, :NVAR-1
A746 +ØØ DB :ANTV
A747 +48A6 DW (:NVAR-1)
A749 SYN :RTN
A749 +Ø3 DB :RTN
< > = <STAT>, <NVAR> <EOS2>#

A74A :SNOTE
A74A :SPOINT
A74A SYN JS, :STAT
A74A +BA DB $80+((((:STAT-*)&$7F) XOR $40 )
A74B SYN CCOM
A74B +12 DB CCOM
A74C SYN :ANTV, AD, :NVAR-1
A74C +ØØ DB :ANTV
A74D +48A6 DW (:NVAR-1)
A74F SYN JS, :EOS2
A74F +E4 DB $80+((((:EOS2-*)&$7F) XOR $40 )
A750 SYN :RTN
A750 +Ø3 DB :RTN
<FS> = <STR>

A751 :FS
A751 SYN :ANTV, AD, :STR-1
A751 +ØØ DB :ANTV
A752 +81A6 DW (:STR-1)
A754 SYN :RTN
A754 +Ø3 DB :RTN
<TEXP> = <EXP>, <EXP>#

A755 :TEXP
A755 SYN :VEXP
A755 +ØE DB :VEXP
A756 SYN CCOM
A756 +12 DB CCOM
A757 SYN :VEXP
A757 +ØE DB :VEXP
A758 SYN :RTN
A758 +Ø3 DB :RTN
< SOUND > = <EXP>, <EXP>, <EXP>, <EXP>, <EOS>#

A759 :SOUND
A759 SYN :VEXP
A759 +ØE DB :VEXP
A75A SYN CCOM
A75A +12 DB CCOM
A75B :SSETCOLOR
A75B SYN :VEXP
A75B +ØE DB :VEXP
A75C SYN CCOM
A75C +12 DB CCOM
< > = <EXP>, <EXP> <EOS>#

A75D :SPOKE
A75D :SPLT
A75D :SPOS
A75D :SDRAWTO
A75D SYN JS, :TEXP
A75D +BB DB $80+((((:TEXP-*)&$7F) XOR $40 )
A75E SYN JS, :EOS2
A75E +Ø5 DB $80+((((:EOS2-*)&$7F) XOR $40 )
A75F SYN :RTN
A75F +Ø3 DB :RTN
<DIM> = <NSML> <EOS>#
A760 :SDIM
A760 :SCOM
A760 SYN JS,:NSML
A760 +EC DB $80+((:(NSML-*)&$7F) XOR $40 )
A761 SYN JS,:EOS2
A761 +D2 DB $80+((:(EOS2-*)&$7F) XOR $40 )
A762 SYN :RTN
A762 +D3 DB :RTN

<ON> = <EXP> <ON1> <EXPL> <EOS>#
A763 :SON SYN :VEXP
A763 +E DB :VEXP
A764 SYN JS,:ON1
A764 +C4 DB $80+((:(ON1-*)&$7F) XOR $40 )
A765 SYN JS,:EXPL
A765 +C7 DB $80+((:(EXPL-*)&$7F) XOR $40 )
A766 SYN JS,:EOS2
A766 +CD DB $80+((:(EOS2-*)&$7F) XOR $40 )
A767 SYN :RTN
A767 +D3 DB :RTN

<ON1> = GOTO | GOSUB#
A768 :ON1 SYN CGTO
A768 +7 DB CGTO
A769 SYN :OR
A769 +2 DB :OR
A76A SYN CGS
A76A +8 DB CGS
A76B SYN :RTN
A76B +D3 DB :RTN

<EXPL> = <EXP> <EXPL1>
A76C :EXPL SYN :VEXP
A76C +E DB :VEXP
A76D SYN JS,:EXPL
A76D +C2 DB $80+((:(EXPL-*)&$7F) XOR $40 )
A76E SYN :RTN
A76E +D3 DB :RTN

<EXPL1> = ,<EXPL> | &#
A76F :EXPL1 SYN CCOM
A76F +C DB CCOM
A770 SYN JS,:EXPL
A770 +BC DB $80+((:(EXPL-*)&$7F) XOR $40 )
A771 SYN :OR
A771 +2 DB :OR
A772 SYN :RTN
A772 +D3 DB :RTN

<EOS2> = CEOS | CCR#
A773 :EOS2
A773 SYN CEOS
A773 +4 DB CEOS
A774 SYN :OR
A774 +2 DB :OR
A775 SYN CCR
A775 +16 DB CCR
A776 SYN :RTN
A776 +D3 DB :RTN

<NSMAT> = <TNVAR> (<EXP> <NMAT2>)
A777 :NSMAT
A777 SYN :ESRT,AD,:TNVAR-1
A777 +D1 DB :ESRT
Source Code

A77A +29A3 DW (:TNVAR-1)
A77A +2B SYN CLPRN, :CHNG, CDLPRN
A77B +OF DB :CHNG
A77C +3F DB CDLPRN
A77D SYN :VEXP
A77D +0E DB :VEXP
A77E SYN :ANTV, AD, :NMAT2-1
A77F +58A6 DW (:NMAT2-1)
A781 SYN CRPRN
A781 +2C DB CRPRN
A782 SYN :OR
A783 DB :OR
A784 +01 DB :ESRT
A784 +2DAA DW (:TSVAR-1)
A785 SYN CLPRN, :CHNG, CDLPRN
A786 +2B DB CLPRN
A787 +OF DB :CHNG
A788 +3F DB CDLPRN
A789 SYN :VEXP
A789 +0E DB :VEXP
A78A SYN CRPRN
A78A +2C DB CRPRN
A78B SYN :RTN
A78B +03 DB :RTN

< NSML > = < NSMAT > < NSML2 > | &

A78C NSML SYN JS, :NSMAT
A78C +AB DB $B0+((( :NSMAT-*)&$7F) XOR $40 )
A78D JS, :NSML2
A78D +C3 DB $B0+((( :NSML2-*)&$7F) XOR $40 )
A78E SYN :OR, :RTN
A78E +02 DB :OR
A78F +03 DB :RTN

< NSML2 > =, < NSML > | &

A790 NSML2 SYN CCOM
A790 +12 DB CCOM
A791 JS, :NSML
A791 +BB DB $B0+((( :NSML-*)&$7F) XOR $40 )
A792 OR, :RTN
A792 +02 DB :OR
A793 +03 DB :RTN

< IF > = < EXP > THEN < IFA > < EOS > #

A794 SIF SYN :VEXP
A794 +0E DB :VEXP
A795 SYN CTTHEN
A795 +1B DB CTTHEN
A796 JS, :IFA
A796 +C3 DB $B0+((( :IFA-*)&$7F) XOR $40 )
A797 JS, :EOS2
A797 +9C DB $B0+((( :EOS2-*)&$7F) XOR $40 )
A798 SYN :RTN
A798 +03 DB :RTN

< IFA > = < TNCON > | < EIF >

A799 :IFA SYN :ESRT, AD, :TNCON-1
A799 +01 DB :ESRT
A79A +FFA3 DW (:TNCON-1)
A79C SYN :OR
A79C +02 DB :OR
A79D +02 DB :ESRT, AD, :EIF-1
A79D +01 DB :ESRT
A79E +D3A2 DW (:EIF-1)
<PR1> = <PEL> | <PSL> <PR2> | &

A7A0 :PR1
SYN JS, :PEL, :OR
A7A0 +C9 DB $80+((:(PEL-*)&$7F) XOR $40 )
A7A1 +O2 DB :OR
A7A2 SYN JS, :PSL
A7A2 +D4 DB $80+(((:PSL-*)&$7F) XOR $40 )
A7A3 SYN JS, :PR2
A7A3 +C3 DB $80+(((PR2-*)&$7F) XOR $40 )
A7A4 SYN :OR
A7A4 +O2 DB :OR
A7A5 SYN :RTN
A7A5 +O3 DB :RTN

<PR2> = <PEL> | &

A7A6 :PR2 SYN JS, :PEL
A7A6 +C3 DB $80+(((PEL-*)&$7F) XOR $40 )
A7A7 SYN :OR
A7A7 +O2 DB :OR
A7A8 SYN :RTN
A7A8 +O3 DB :RTN

<PEL> = <PES> <PELA> #

A7A9 :PEL SYN JS, :PES
A7A9 +C3 DB $80+(((PES-*)&$7F) XOR $40 )
A7AA SYN JS, :PELA
A7AA +C8 DB $80+(((PELA-*)&$7F) XOR $40 )
A7AB SYN :RTN
A7AB +O3 DB :RTN

<PES> = <EXP> | <STR>

A7AC :PES SYN :VEXP
A7AC +0E DB :VEXP
A7AD SYN :OR
A7AD +O2 DB :OR
A7AE SYN :ANTV, AD, :STR-1
A7AE +00 DB :ANTV
A7AF +B1A6 DW ( :STR-1 )
A7B1 SYN :RTN
A7B1 +O3 DB :RTN

<PELA> = <PSL> <PEL> | &

A7B2 :PELA SYN JS, :PSL
A7B2 +C4 DB $80+(((PSL-*)&$7F) XOR $40 )
A7B3 SYN JS, :PR2
A7B3 +O3 DB $80+(((PR2-*)&$7F) XOR $40 )
A7B4 SYN :OR
A7B4 +O2 DB :OR
A7B5 SYN :RTN
A7B5 +O3 DB :RTN

<PSL> = <PS> <PSLA> #

A7B6 :PSL SYN JS, :PS
A7B6 +C6 DB $80+(((PS-*)&$7F) XOR $40 )
A7B7 SYN JS, :PSLA
A7B7 +C2 DB $80+(((PSLA-*)&$7F) XOR $40 )
A7B8 SYN :RTN
A7B8 +O3 DB :RTN

<PSLA> = <PSL> | &

A7B9 :PSLA SYN JS, :PSL
A7B9 +BD DB $80+(((PSL-*)&$7F) XOR $40 )
A7BA SYN :OR

175
Source Code

A7BA +02 DB :OR
A7BB SYN :RTN
A7BB +03 DB :RTN

<PS> =, |, #

A7BC :PS SYN CCOM
A7BC +12 DB CCOM
A7BD SYN :OR
A7BD +02 DB :OR
A7BE SYN CSC
A7BE +15 DB CSC
A7BF SYN :RTN
A7BF +03 DB :RTN

<L1> = <EXP> <L2> | &#

A7C0 :L1 SYN :VEXP
A7C9 +0E DB :VEXP
A7C1 SYN JS, :L2
A7C1 +C3 DB $80+(((L2-*)&$7F) XOR $40)
A7C2 SYN :OR
A7C2 +02 DB :OR
A7C3 SYN :RTN
A7C3 +03 DB :RTN

<L2> =, <EXP> | &#

A7C4 :L2 SYN CCOM
A7C4 +12 DB CCOM
A7C5 SYN :VEXP
A7C6 SYN :VEXP
A7C6 +02 DB :OR
A7C7 SYN :RTN
A7C7 +03 DB :RTN

<REM> = <EREM>

A7C8 :SREM SYN :ERT,AD,:EREM-1
A7C8 +01 DB :ERT
A7C9 +DFA2 DW (:EREM-1)

<SDATA> = <EDATA>

A7CB :SDATA SYN :ESRT,AD,:EDATA-1
A7CB +01 DB :ESRT
A7CC +DFA2 DW (:EDATA-1)

<NFSP> = ASC | VAL | LEN#

A7CE :NFSP SYN CASC, :OR
A7CE +40 DB CASC
A7CE +02 DB :OR
A7D0 SYN CVAL, :OR
A7D0 +41 DB CVAL
A7D1 +02 DB :OR
A7D2 SYN CADR, :OR
A7D2 +43 DB CADR
A7D3 +02 DB :OR
A7D4 SYN CLEN
A7D4 +42 DB CLEN
A7D5 SYN :RTN
A7D5 +03 DB :RTN

<SFNP> = STR | CHR#

:SFNP SYN CSTR,:OR
A7D6 +3D DB CSTR
A7D7 +Ø2 DB :OR
A7DB SYN CCHR
A7DB +3E DB CCHR
A7D9 SYN :RTN
A7D9 +03 DB :RTN

<PUSR> = <EXP> <PUSR1> #

:PUSR SYN :VEXP
A7DA +ØE DB :VEXP
A7DB SYN JS,:PUSR1
A7DB +C2 DB $8Ø+((,:PUSR1-*)&$7F) XOR $4Ø )
A7DC SYN :RTN
A7DC +03 DB :RTN

<PUSR1> =,<PUSR> | &#

:PUUSR1 SYN CCOM,:CHNG,CACOM
A7DD +12 DB CCOM
A7DE +ØF DB :CHNG
A7DF +3C DB CACOM
A7EØ SYN JS,:PUSR
A7EØ +BA DB $8Ø+((,:PUSR-*)&$7F) XOR $4Ø )
A7E1 SYN :OR
A7E1 +02 DB :OR
A7E2 SYN :RTN
A7E2 +03 DB :RTN

OPNTAB — Operator Name Table

A7E3 = 000F OPNTAB C SET $8F ;FIRST ENTRY VALUE=$10
  = 0010
  = 0010 CDQ EQU C
A7E3 82 DB $82 ;DOUBLE QUOTE
  = 0011 C SET C+1
  = 0011 CSOE EQU C
A7E4 80 DB $80 ; DUMMY FOR SOE
  = 0012 C SET C+1
  = 0012 CCOM EQU C
A7E5 AC DC ','
  = 0013 C SET C+1
  = 0013 CDOL EQU C
A7E6 A4 DC '$'
  = 0014 C SET C+1
  = 0014 CROS EQU C
A7E7 BA DC ','
  = 0015 C SET C+1
  = 0015 CSC EQU C
A7E8 BB DC ','
  = 0016 C SET C+1
  = 0016 CCR EQU C ;CARRIAGE RETURN
A7E9 9B DB CR
  = 0017 C SET C+1
  = 0017 CGTO EQU C
A7EA 474F54CF DC 'GOTO'
Source Code

```
= 0018  C SET C+1
= 0018  CGS EQU C
A7EE 474F5355C2 DC 'GOSUB'
= 0019  C SET C+1
= 0019  CTO EQU C
A7F3 54CF DC 'TO'
= 001A  C SET C+1
= 001A  CSTEP EQU C
A7F5 535445D0 DC 'STEP'
= 001B  C SET C+1
= 001B  CTEN EQU C
A7F9 544B45CE DC 'THEN'
= 001C  C SET C+1
= 001C  CPND EQU C
A7FD A3 DC '#'
= 001D  CSROP EQU C+1 ; START OF REAL OPs
= 001D  C SET C+1
= 001D  CLE EQU C
A7FE 3CBD DC '<='
= 001E  C SET C+1
= 001E  CNE EQU C
A800 3CBE DC '>'
= 001F  C SET C+1
= 001F  CGE EQU C
A802 3EBD DC '>='
= 0020  C SET C+1
= 0020  CLT EQU C
A804 BC DC '<'
= 0021  C SET C+1
= 0021  CGT EQU C
A805 BE DC '>'
= 0022  C SET C+1
= 0022  CEQ EQU C
A806 BD DC '='
= 0023  C SET C+1
= 0023  CEXP EQU C
A807 DE DB $58+$80 ; UP ARROW FOR EXP
= 0024  C SET C+1
= 0024  CMUL EQU C
A808 AA DC '+'
= 0025  C SET C+1
= 0025  CPLUS EQU C
A809 AB DC '-'
= 0026  C SET C+1
= 0026  CMINUS EQU C
A80A AD DC '/'
= 0027  C SET C+1
= 0027  CDIV EQU C
A80B AF DC 'NOT'
= 0028  C SET C+1
= 0028  CNOT EQU C
A80C 4E4FD4 DC 'NOT'
```

178
Source Code

Source Code

= 0029 C SET C+1
= 0029 COR EQU C
A80F 4FD2 DC 'OR'

= 002A C SET C+1
= 002A CAND EQU C
A811 414EC4 DC 'AND'

= 002B C SET C+1
= 002B CLPRN EQU C
A814 A8 DC '('

= 002C C SET C+1
= 002C CRPRN EQU C
A815 A9 DC ')',

; THE FOLLOWING ENTRIES ARE COMPRISED OF CHARACTERS
; SIMILAR TO SOME OF THOSE ABOVE BUT HAVE
; DIFFERENT SYNTACTICAL OR SEMANTICAL MEANING
;

= 002D C SET C+1
= 002D CAASN EQU C ; ARITHMETIC ASSIGNMENT
A816 BD DC '=,'

= 002E C SET C+1
= 002E CSASN EQU C ; STRING OPS
A817 BD DC '=,'

= 002F C SET C+1
= 002F CSLE EQU C
A818 3CBD DC '<= '

= 0030 C SET C+1
= 0030 CSNE EQU C
A81A 3CBE DC '<>'

= 0031 C SET C+1
= 0031 CSGE EQU C
A81C 3EBD DC '=>'

= 0032 C SET C+1
= 0032 CSLT EQU C
A81E BC DC '<'

= 0033 C SET C+1
= 0033 CSGT EQU C
A81F BE DC '>

= 0034 C SET C+1
= 0034 CSEQ EQU C
A820 BD DC '=

= 0035 C SET C+1
= 0035 CPLUS EQU C ; UNARY PLUS
A821 AB DC '+

= 0036 C SET C+1
= 0036 CUMINUS EQU C ; UNARY MINUS
A822 AD DC '-

= 0037 C SET C+1
= 0037 CSLPRN EQU C ; STRING LEFT PAREN
A823 A8 DC '('

= 0038 C SET C+1
= 0038 CALPRN EQU C ; ARRAY LEFT PAREN
A824 $80 DB $80 ; DOES NOT PRINT

= 0039 C SET C+1
= 0039 CDLPRN EQU C ; DIM LEFT PAREN
Source Code

A825 80                   DB     $80                   ; DOES NOT PRINT
    = 003A                  C      SET C+1
    = 003A                  CFLPRN EQU C  ; FUNCTION LEFT PAREN
A826 A8                   DC     '('
    = 003B                  C      SET C+1
    = 003B                  CDSLPR EQU C
A827 A8                   DC     '('
    = 003C                  C      SET C+1
    = 003C                  CACOM EQU C  ; ARRAY COMMA
A828 AC                   DC     ',,'

Function Name Table

; PART OF ONTAB
;
A829                  
    = 003D                  C      SET C+1
    = 003D                  CFFUN EQU C  ; FIRST FUNCTION CODE
A829                  535452A4 DC 'STRS' 
    = 003E                  C      SET C+1
    = 003E                  CCHR EQU C
A82D 434852A4 DC 'CHR$'
    = 003F                  C      SET C+1
    = 003F                  CUSR EQU C  ; USR FUNCTION CODE
A831 5553D2 DC 'USR' 
    = 0040                  C      SET C+1
    = 0040                  CASC EQU C
A834 4153C3 DC 'ASC' 
    = 0041                  C      SET C+1
    = 0041                  CVAL EQU C
A837 5641CC DC 'VAL' 
    = 0042                  C      SET C+1
    = 0042                  CLEN EQU C
A83A 4C45CE DC 'LEN' 
    = 0043                  C      SET C+1
    = 0043                  CADR EQU C
A83D 4144D2 DC 'ADR' 
    = 0044                  C      SET C+1
    = 0044                  CNFNP EQU C
A840 4154CE DC 'ATN' 
A843 434FD3 DC 'COS' 
A846 504545CB DC 'PEEK' 
A84A 5349CE DC 'SIN' 
A84D 524EC4 DC 'RND' 
A850 4652C5 DC 'FRE' 
A853 4558D0 DC 'EXP' 
A856 4C4FC7 DC 'LOG' 
A859 434C4PC7 DC 'CLOG' 
A85D 5351D2 DC 'SQRT' 
A860 5347CE DC 'SGN' 
A863 4142D3 DC 'ABS' 
A866 494ED4 DC 'INT' 
A869 564144444C DC 'PADDLE' 
    ; C5
A86F 53544943CB DC 'STICK' 
A874 50545249C7 DC 'PTRIG' 
A879 53545249C7 DC 'STRIG' 
A87E 00                   DB     $00
    ; END OF OPNTAB & FNTAB
Memory Manager

A87F

LOCAL

; MEMORY MANAGEMENT CONSISTS OF EXPANDING AND
; CONTRACTING TO INFORMATION AREA POINTED TO
; BY THE ZERO PAGE POINTER TABLES. ROUTINES
; MODIFY THE ADDRESS IN THE TABLES AND
; MOVE DATA AS REQUIRED. THE TWO FUNDAMENTAL
; ROUTINES ARE 'EXPAND' AND 'CONTRACT'

EXPAND

; X = ZERO PAGE ADDRESS OF TABLE AT WHICH
; EXPANSION IS TO START
; Y = EXPANSION SIZE IN BYTES [LOW]
; A = EXPANSION SIZE IN BYTES [HIGH]
; EXPLow - FOR EXPANSION < 256 BYTES
; SETS A = 0

A87F A900

EXPLow LDA #$0

A881

EXPAND LDA #'

A883 85A5

STA ECSIZE+1

A885 38

SEC

A886 A500

LDA MEMTOP

A888 65A4

ADC ECSIZE

A889 A8

TAX

A88B A591

LDA MEMTOP+1

A88D 65A5

ADC ECSIZE+1

A88F CDE602

CMP HIMEM+1

A892 900C A8A0

BCC :EXP2

A894 D007 A89D

BNE :EXP1

A896 CE502

CPY HIMEM

A899 9005 A8A0

BCC :EXP2

A89B F083 A8A0

BEQ :EXP2

A89D 4C3C99

:EXP3 JMP MEMFULL

:EXP2

A8A0

SEC

A8A1 A500

LDA MEMTOP

A8A3 F500

SBC $X

A8A5 85A2

STA MVLNG

A8A7 A591

LDA MEMTOP+1

A8A9 F501

SBC $1,X

A8AB 85A3

STA MVLNG+1

A8AD 18

CLC

A8AE 7501

ADC $1,X

A8B0 859A

STA MVFA+1

A8B2 B500

LDA $0,X

A8B4 8599

STA MVFA

A8B6 8597

STA SVESA

A8B8 65A4

ADC ECSIZE

A8BA 859B

STA MVTA

A8BC B501

LDA $1,X

A8BE 8598

STA SVESA+1

A8C0 65A5

ADC ECSIZE+1

A8C2 65A3

ADC MVLNG+1

A8C4 859C

STA MVTA+1

A8C6

:EXP3

181
Source Code

A8C6 B500 LDA 0,X ; ADD ECSIZE TO
A8CB 65A4 ADC ECSIZE ; ALL TABLE ENTRIES
A8CA 9500 STA 0,X ; FROM EXPAND AT ADR
A8CC B501 LDA 1,X ; TO HIMEM
A8CE 65A5 ADC ECSIZE+1
A8D0 9501 STA 1,X

A8D2 E8 INX
A8D3 E8 INX
A8D4 E092 CPX #$MEMTOP+2
A8D6 9B2E "ABC6 B500 ADC ECSIZE TO ALL TABLE ENTRIES
A8D8 85F STA APHM+1 ; SET NEW AP
A8DA A590 LDA MEMTOP ; HI MEM TO
A8DC 850E STA APHM ; MEMTOP

A8DE A6A3 LDX MVLNG+1 ; X = MVLNG[H]
A8E0 E8 INX PLUS ONE
A8E1 A4A2 LDY MVLNG ; Y = MVLNG[L]
A8E3 D000 "A8F0 BNE :EXP6 ; TEST ZERO LENGTH
A8E5 F010 "A8F7 BEQ :EXP7 ; BR IF LOW = 0

A8E7 88 :EXP4 DEY ; DEC MVLNG[L]
A8E8 C69A DEC MVFA+1 ; DEC MVFA[H]
A8EA C69C DEC MVT[H]

A8EC B199 :EXP5 LDA [MVFA],Y ; MVFA BYTE
A8EE 9198 STA [MVT],Y ; TO MVT
A8F0 88 :EXP6 DEY ; DEC COUNT LOW
A8F1 D0F9 "A8EC BNE :EXP5 ; BR IF NOT ZERO

A8F3 B199 LDA [MVFA],Y ; MOVE THE ZERO BYTE
A8F5 919B STA [MVT],Y

A8F7 :EXP7
A8F7 CA DEX ; IF MVLNG[H] IS NOT
A8F8 D0ED "A8F7 BNE :EXP4 ; ZERO THEN MOVE 256 MORE

A8FA 60 RTS DONE ; ELSE

CONTRACT

; X = ZERO PAGE ADR OF TABLE AT WHICH
; CONTRACTION WILL START
; Y = CONTRACT SIZE IN BYTES [LOW]
; A = CONTRACT SIZE IN BYTES [HI]
; CONTLOW
; SETS A = 0

A8FB A900 CONTLOW LDA #0

A8FD CONTRACT

A8FD 84A4 STY ECSIZE ; SAVE CONTRACT SIZE
A8FF 85A5 STA ECSIZE+1

A901 38 SEC MEMTOP
A902 A590 LDA 0,X ; MVLNG[L] = $100-
A904 F500 SBC 0,X ; [MEMTOP[L]] - CON AT
A906 49FF EOR $FF VALUE [L]
A908 A8 TAY ; THIS MAKES START Y AT
A909 CB INY ; MOVE HAVE A 2'S COMPLEMENT
A90A 84A2 STY MVLNG ; REMAINDER IN IT

A90C A591 LDA MEMTOP+1 ; FORM MOVE LENGTH[HIGH]
A90E F501 SBC l,X
A910 85A3 STA MVLNG+1

A912 B500 LDA 0,X ; FORM MOVE FROM ADR [MVFA]
A914 E5A2 SBC MVLNG ; MVFA = CON AT VALUE
A916 B599 STA MVFA ; MINUS MVLNG[L]
A918 B501 LDA 1,X ; DURING MOVE MVLNG[L]
A91A E900 SBC #0 ; WILL BE ADDED BACK INTO
A91C 859A STA MVFA+1 ; MVFA IN [IND], Y INST
A91E 869B STX MVTA ; TEMP SAVE OF CON AT DISPL
A920 30 :CONT1 SEC ; SUBTRACT ECSIZE FROM
A921 B500 LDA 0,X ; ALL TABLE ENTRY FROM
A923 E5A4 SBC ECSIZE ; CON AT ADR TO HIMEM
A925 9500 STA 0,X
A927 B501 LDA 1,X
A929 E5A5 SBC ECSIZE+1
A92B 9501 STA 1,X
A92D E8 INX
A92E E0 INX
A92F E092 CPX #MEMTOP+2
A931 90ED A A920 BCC :CONT1
A933 850F STA APHM+1 ; SET NEW APL
A935 A590 LDA MEMTOP ; HI MEM TO
A937 850E STA APHM ; MEMTOP
A939 A69B LDX MVTA
A93B B500 LDA 0,X ; FORM MOVE TO ADR [MVTA]
A93D E5A2 SBC MVLNG ; MVTA = NEW CON AT VALUE
A93F 859B STA MVTA ; MINUS MVLNG [L]
A941 9501 LDA 1,X ; DURING MOVE MVLNG[L]
A943 E900 SBC #0 ; WILL BE ADDED BACK INTO
A945 859C STA MVTA+1 ; MVTA IN [INO], Y INST

A947 FMOVER
A949 A6A3 LDX MVLNG+1 ; GET MOVE LENGTH HIGH
A949 E8 INX ; INC SO MOVE CAN BNE
A94A AA2 LDY MVLNG ; GET MOVE LENGTH LOW
A94C D006 A954 BNE :CONT2 ; IF NOT ZERO GO
A94E F00B A95B BEQ :CONT4 ; BR IF LOW = 0
A950 E69A :CONT3 INC MVFA+1 ; INC MVFA[H]
A952 E69C INC MVTA+1 ; INC MVTA[H]
A954 B199 :CONT2 LDA [MVFA], Y ; GET MOVE FROM BYTE
A956 919B STA [MVTA], Y ; SET MOVE TO BYTE
A958 C8 INY ; INCREMENT COUNT LOW
A959 D0F9 A954 BNE :CONT2 ; BR IF NOT ZERO
A95B :CONT4
A95B CA DEX ; DECREMENT COUNT HIGH
A95C D0F2 A950 BNE :CONT3 ; BR IF NOT ZERO
A95E 60 RTS ; ELSE DONE

Execute Control

A95F LOCAL

EXECNL — Execute Next Line

; START PROGRAM EXECUTOR
; EXECNL EXECNS — Execute Next Statement
;
A95F 201BB8 JSR SETLN1 ; SET UP LIN & NXT STMT

EXECNS — Execute Next Statement

A962 EXECNS
A962 20F4A9 JSR TSTBRK ; TEST BREAK
A965 D035 A99C BNE :EXBRK ; BR IF BREAK
A967 A4A7 LDY NXTSTD ; GET PTR TO NEXT STMT L
A969 C49F CPY LNGTH ; AT END OF LINE
A96B B01C A989 BCS :EXEOL ; BR IF EOL
GETSTMT — Get Statement in Statement Table

SEARCH FOR STMT THAT HAS TSLNUM
SET STMCUR TO POINT TO IT IF FOUND
OR TO WHERE IT WOULD GO IF NOT FOUND
CARRY SET IF NOT FOUND

A9A2 GETSTMT

SAVE CURRENT LINE ADDR

A9A2 A50A LDA STMCUR
A9A4 85BE STA SAVCUR
A9A6 A58B LDA STMCUR+1
A9A8 85BF STA SAVCUR+1
A9AA A589 LDA STMTAB+1
A9AC A488 LDX STMTAB
A9AE 85B8 STA STMCUR+1
A9B0 84A8 STY STM CUR

A9B2 A001 1GS2 LDY #1
A9B4 B18A LDA [STMCUR],Y
A9B6 C5A1 CMP TSLNUM+1
A9B8 900D "A9C7 BCC :GS3
A9BA D00A "A9C6 BNE :GSR T1
A9BC 86 DEY :GS3
A9BD B18A LDA [STMCUR],Y
A9BF C5A0 CMP TSLNUM
A9C1 9004 "A9C7 BCC :GS3
A9C3 D001 "A9C6 BNE :GSR T1
A9C5 19 CLC
A9C6 60 RTS

A9C7 20DDA9 1GS3 JSR GETLL

184
A9CA 20D0A9  JSR  GNXTL  
A9CD 4CB2A9  JMP  'GS2  

;  
A9D0  GXTNL  
A9D0  10  CLC  
A9D1  658A  ADC  STMCUR  ;ADD LENGTH TO STMCUR  
A9D3  858A  STA  STMCUR  
A9D5  A8  TAY  
A9D6  A58B  LDA  STMCUR+1  
A9D8  6900  ADC  $0  
A9DA  858B  STA  STMCUR+1  
A9DC  60  RTS  
A9DD  A002  GETLL  LDY  #2  
A9DF  B18A  LDA  [STMCUR],Y  
A9E1  60  RTS  

TENDST — Test End of Statement Table  
A9E2  TENDST  
A9E2  A001  LDY  #1  ; INDEX TO CNO ['I']  
A9E4  858B  STA  STMCUR  
A9E6  60  RTS  
A9E7  XREM  
A9E7  XDATA  
A9E7  60  TESTRTS RTS  

XBYE — Execute BYE  
A9E8  XBYE  
A9E8  2041BD  JSR  CLSALL  ; CLOSE 1-7  
A9E9  4C71E4  JMP  BYELOC  ; EXIT  

XDOS — Execute DOS  
A9EE  XDOS  
A9EE  2041BD  JSR  CLSALL  ; CLOSE 1-7  
A9F1  6C0A00  JMP  [DOSLOC]  ; GO TO DOS  

TSTBRK — Test for Break  
A9F4  TSTBRK  
A9F4  A000  LDY  #0  
A9F6  A511  LDA  BRKBYTE  ; LOAD BREAKBYTE  
A9F8  D004 A9FE  BNE  :TB2  
A9FA  A0FF  LDY  #$FF  
A9FC  8411  STY  BRKBYTE  
A9FE  90  :TB2  TYA  ; SET COND CODE  
A9FF  60  RTS  ; DONE  

Statement Execution Table  
;STETAB—STATEMENT EXECUTION TABLE  
;    -CONTAINS STMNT EXECUTIONADR  
;    -MUST BE IN SAME ORDER AS STNTAB  
;  
AA00  STETAB  
AA00  FDB  XREM-1  
AA00  +A9E6  DW  REV (XREM-1)  
AA02  FDB  XDATA-1  
AA02  +A9E6  DW  REV (XDATA-1)  
   = 0001  CDATA  EQU  (+STETAB)/2-1  
AA04  FDB  XINPUT-1  
AA04  +B315  DW  REV (XINPUT-1)  
AA06  FDB  XCOLOR-1  
AA06  +BA28  DW  REV (XCOLOR-1)  
AA08  FDB  XLIST-1  
AA08  +B402  DW  REV (XLIST-1)  
   = 0004  CLIST  EQU  (+-STETAB)/2-1  

185
Source Code

AA0A FDB XENTER-1
AA0A +BACA DW REV (XENTER-1)
AA0C FDB XLET-1
AA0C +AADF DW REV (XLET-1)
AA0E FDB XIF-1
AA0E +B777 DW REV (XIF-1)
AA10 FDB XFOR-1
AA10 +B64A DW REV (XFOR-1)
= 000B CPOR EQU
AA12 FDB XNEXT-1
AA12 +B6CE DW REV (XNEXT-1)
AA14 FDB XGOTO-1
AA14 +B6A2 DW REV (XGOTO-1)
AA16 FDB XGOTO-1
AA16 +B6A2 DW REV (XGOTO-1)
AA18 FDB XGOSUB-1
AA18 +B69F DW REV (XGOSUB-1)
= 000C CGOSUB EQU
AA1A FDB XTRAP-1
AA1A +B7E0 DW REV (XTRAP-1)
AA1C FDB XBYE-1
AA1C +A9E7 DW REV (XBYE-1)
AA1E FDB XCONT-1
AA1E +B7BD DW REV (XCONT-1)
AA20 FDB XCOM-1
AA20 +B1D8 DW REV (XCOM-1)
AA22 FDB XCLOSE-1
AA22 +BC1A DW REV (XCLOSE-1)
AA24 FDB XCLR-1
AA24 +B765 DW REV (XCLR-1)
AA26 FDB XDEG-1
AA26 +B260 DW REV (XDEG-1)
AA28 FDB XDIM-1
AA28 +B1D8 DW REV (XDIM-1)
AA2A FDB XEND-1
AA2A +B78C DW REV (XEND-1)
AA2C FDB XNEW-1
AA2C +BA0B DW REV (XNEW-1)
AA2E FDB XOPE-1
AA2E +BBEA DW REV (XOPE-1)
AA30 FDB XLOAD-1
AA30 +BAFA DW REV (XLOAD-1)
AA32 FDB XSAVE-1
AA32 +BB5C DW REV (XSAVE-1)
AA34 FDB XSTATUS-1
AA34 +BC27 DW REV (XSTATUS-1)
AA36 FDB XNOTE-1
AA36 +BC35 DW REV (XNOTE-1)
AA38 FDB XPOINT-1
AA38 +BC4C DW REV (XPOINT-1)
AA3A FDB XXIO-1
AA3A +BB6C DW REV (XXIO-1)
AA3C FDB XON-1
AA3C +B7EC DW REV (XON-1)
= 001E CON EQU
AA3E FDB XPOKE-1
AA3E +B24B DW REV (XPOKE-1)
AA40 FDB XPRINT-1
AA40 +B3B5 DW REV (XPRINT-1)
AA42 FDB XRAD-1
AA42 +B265 DW REV (XRAD-1)
AA44 FDB XRAD-1
AA44 +B282 DW REV (XRAD-1)
= 0022 CREAD EQU
AA46 FDB XREST-1
AA46 +B26A DW REV (XREST-1)
AA48 FDB XRTN-1
AA48 +B71B DW REV (XRTN-1)
AA4A FDB XRUN-1
AA4A +B74C DW REV (XRUN-1)
AA4C FDB XSTOP-1

186
### Source Code

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AA4C +B792</td>
<td><strong>DW</strong> REV (XSTOP-1)</td>
<td>Source Code</td>
</tr>
<tr>
<td>AA4E +B840</td>
<td><strong>FDB</strong> REV (XPOP-1)</td>
<td>Source Code</td>
</tr>
<tr>
<td>AA50 +B3B5</td>
<td><strong>DW</strong> REV (XPRI NT-1)</td>
<td>Source Code</td>
</tr>
<tr>
<td>AA52 +BC7E</td>
<td><strong>FDB</strong> XGET-1</td>
<td>Source Code</td>
</tr>
<tr>
<td>AA54 +BC71</td>
<td><strong>DW</strong> REV (XPUT-1)</td>
<td>Source Code</td>
</tr>
<tr>
<td>AA56 +BA4F</td>
<td><strong>FDB</strong> REV (XGR-1)</td>
<td>Source Code</td>
</tr>
<tr>
<td>AA58 +BA75</td>
<td><strong>DW</strong> REV (XPRINT-1)</td>
<td>Source Code</td>
</tr>
<tr>
<td>AA5A +BA15</td>
<td><strong>FDB</strong> XPOS-1</td>
<td>Source Code</td>
</tr>
<tr>
<td>AA5C +A9ED</td>
<td><strong>DW</strong> REV (XDOS-1)</td>
<td>Source Code</td>
</tr>
<tr>
<td>AA5E +BA30</td>
<td><strong>FDB</strong> XRDAWTO-1</td>
<td>Source Code</td>
</tr>
<tr>
<td>AA60 +BA36</td>
<td><strong>FDB</strong> REV (XSETCOLOR-1)</td>
<td>Source Code</td>
</tr>
<tr>
<td>AA62 +B96E</td>
<td><strong>DW</strong> REV (XSETCOLOR-1)</td>
<td>Source Code</td>
</tr>
<tr>
<td>AA64 +BC94</td>
<td><strong>DW</strong> REV (XLOCATE-1)</td>
<td>Source Code</td>
</tr>
<tr>
<td>AA66 +B9DC</td>
<td><strong>FDB</strong> XSOUND-1</td>
<td>Source Code</td>
</tr>
<tr>
<td>AA68 +BC63</td>
<td><strong>FDB</strong> REV (XLPRINT-1)</td>
<td>Source Code</td>
</tr>
<tr>
<td>AA6A +BBA3</td>
<td><strong>FDB</strong> XCSAVE-1</td>
<td>Source Code</td>
</tr>
<tr>
<td>AA6C +B0A3</td>
<td><strong>FDB</strong> REV (XCSAVE-1)</td>
<td>Source Code</td>
</tr>
<tr>
<td>AA6E +B91D</td>
<td><strong>FDB</strong> XPLE-1</td>
<td>Source Code</td>
</tr>
<tr>
<td>= 0036</td>
<td><strong>EQU</strong> (*-STETAB)/2-1</td>
<td>Source Code</td>
</tr>
<tr>
<td>AA6E +B91D</td>
<td><strong>DW</strong> REV (XERR-1)</td>
<td>Source Code</td>
</tr>
<tr>
<td>= 0037</td>
<td><strong>EQU</strong> (*-STETAB)/2-1</td>
<td>Source Code</td>
</tr>
</tbody>
</table>

### Operator Execution Table

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AA70</td>
<td><strong>FDB</strong> XPLE-1</td>
<td>Operator Execution Table</td>
</tr>
<tr>
<td>AA70 +ACB4</td>
<td><strong>DW</strong> REV (XPLE-1)</td>
<td>Operator Execution Table</td>
</tr>
<tr>
<td>AA72</td>
<td><strong>FDB</strong> XPNE-1</td>
<td>Operator Execution Table</td>
</tr>
<tr>
<td>AA74 +ACBD</td>
<td><strong>FDB</strong> XPGE-1</td>
<td>Operator Execution Table</td>
</tr>
<tr>
<td>AA74 +ACD4</td>
<td><strong>DW</strong> REV (XPGE-1)</td>
<td>Operator Execution Table</td>
</tr>
<tr>
<td>AA76</td>
<td><strong>FDB</strong> XPLT-1</td>
<td>Operator Execution Table</td>
</tr>
<tr>
<td>AA76 +ACC4</td>
<td><strong>DW</strong> REV (XPLT-1)</td>
<td>Operator Execution Table</td>
</tr>
<tr>
<td>AA78 +ACC8</td>
<td><strong>DW</strong> REV (XPGT-1)</td>
<td>Operator Execution Table</td>
</tr>
<tr>
<td>AA7A</td>
<td><strong>FDB</strong> XPGEQ-1</td>
<td>Operator Execution Table</td>
</tr>
<tr>
<td>AA7A +ACDB</td>
<td><strong>DW</strong> REV (XPGEQ-1)</td>
<td>Operator Execution Table</td>
</tr>
<tr>
<td>AA7C</td>
<td><strong>FDB</strong> XPPOWER-1</td>
<td>Operator Execution Table</td>
</tr>
<tr>
<td>AA7C +B164</td>
<td><strong>DW</strong> REV (XPPOWER-1)</td>
<td>Operator Execution Table</td>
</tr>
<tr>
<td>AA7E</td>
<td><strong>FDB</strong> XPMLUS-1</td>
<td>Operator Execution Table</td>
</tr>
<tr>
<td>AA7E +AC95</td>
<td><strong>DW</strong> REV (XPMLUS-1)</td>
<td>Operator Execution Table</td>
</tr>
<tr>
<td>AA80</td>
<td><strong>FDB</strong> XPMLUS-1</td>
<td>Operator Execution Table</td>
</tr>
<tr>
<td>AA80 +AC83</td>
<td><strong>DW</strong> REV (XPMLUS-1)</td>
<td>Operator Execution Table</td>
</tr>
<tr>
<td>AA82</td>
<td><strong>FDB</strong> XPMLUS-1</td>
<td>Operator Execution Table</td>
</tr>
<tr>
<td>AA82 +AC8C</td>
<td><strong>DW</strong> REV (XPMLUS-1)</td>
<td>Operator Execution Table</td>
</tr>
<tr>
<td>AA84</td>
<td><strong>FDB</strong> XPDIV-1</td>
<td>Operator Execution Table</td>
</tr>
<tr>
<td>AA84 +AC9E</td>
<td><strong>DW</strong> REV (XPDIV-1)</td>
<td>Operator Execution Table</td>
</tr>
<tr>
<td>AA86</td>
<td><strong>FDB</strong> XPNOT-1</td>
<td>Operator Execution Table</td>
</tr>
<tr>
<td>AA86 +ACF8</td>
<td><strong>DW</strong> REV (XPNOT-1)</td>
<td>Operator Execution Table</td>
</tr>
<tr>
<td>AA88</td>
<td><strong>FDB</strong> XPOR-1</td>
<td>Operator Execution Table</td>
</tr>
<tr>
<td>AA88 +ACED</td>
<td><strong>DW</strong> REV (XPOR-1)</td>
<td>Operator Execution Table</td>
</tr>
</tbody>
</table>
Source Code

```assembly
AA8A FDB XPAND-1
AA8A +ACE2 DW REV (XPAND-1)
AA8C FDB XPPLRN-1
AA8C +AB1E DW REV (XPPLRN-1)
AA8E FDB XPRPRN-1
AA8E +AD7A DW REV (XPRPRN-1)
AA90 FDB XPAASN-1
AA90 +AD5E DW REV (XPAASN-1)
AA92 FDB XSAASN-1
AA92 +AEA2 DW REV (XSAASN-1)
AA94 FDB XPSLE-1
AA94 +ACB4 DW REV (XPSLE-1)
AA96 FDB XPSNE-1
AA96 +ACBD DW REV (XPSNE-1)
AA98 FDB XPSGE-1
AA98 +ACD4 DW REV (XPSGE-1)
AA9A FDB XPSLT-1
AA9A +ACC4 DW REV (XPSLT-1)
AA9C FDB XPSGT-1
AA9C +ACCB DW REV (XPSGT-1)
AA9E FDB XPEQ-1
AA9E +ACDB DW REV (XPEQ-1)
AAA0 FDB XPULPN-1
AAA0 +ACB3 DW REV (XPULPN-1)
AAA2 FDB XPMNUS-1
AAA2 +ACA7 DW REV (XPMNUS-1)
AAA4 FDB XSPLRN-1
AAA4 +AE25 DW REV (XSPLRN-1)
AAA6 FDB XPALPN-1
AAA6 +AD85 DW REV (XPALPN-1)
AAA8 FDB XPDLPN-1
AAA8 +AD81 DW REV (XPDLPN-1)
AAA9 FDB XPPLPN-1
AAA9 +AD7A DW REV (XPPLPN-1)
AAC0 FDB XP SLP-1
AAC0 +AD81 DW REV (XP SLP-1)
AAC1 FDB XPACOM-1
AAC1 +AD78 DW REV (XPACOM-1)

AA80 FDB XPSTR-1
AA80 +B048 DW REV (XPSTR-1)
AA82 FDB XPCHR-1
AA82 +B066 DW REV (XPCHR-1)
AA84 FDB XPUSR-1
AA84 +B089 DW REV (XPUSR-1)
AA86 FDB XPASC-1
AA86 +B011 DW REV (XPASC-1)
AA88 FDB XPVAL-1
AA88 +AFFF DW REV (XPVAL-1)
AA8A FDB XPLEN-1
AA8A +AFC9 DW REV (XPLEN-1)
AA8C FDB XPADR-1
AA8C +B01B DW REV (XPADR-1)
AA8E FDB XPATN-1
AA8E +B12E DW REV (XPATN-1)
AAC0 FDB XPCOS-1
AAC0 +B124 DW REV (XPCOS-1)
AAC2 FDB XPPEEK-1
AAC2 +AFE0 DW REV (XPPEEK-1)
AAC4 FDB XPSIN-1
AAC4 +B11A DW REV (XPSIN-1)
AAC6 FDB XP RND-1
AAC6 +B08A DW REV (XP RND-1)
AAC8 FDB XPFRE-1
AAC8 +APEA DW REV (XP FRE-1)
AAC A FDB XPEXP-1
AAC A +B14C DW REV (XPEXP-1)
AAC C FDB XPLLOG-1
AAC C +B13B DW REV (XPLLOG-1)
AAC E FDB XPL10-1
AAC E +B142 DW REV (XPL10-1)
```

188
AADØ  FDB  XPSQR-1
AADØ +B156  DW  REV (XPSQR-1)
AAD2  FDB  XPSGN-1
AAD2 +AD18  DW  REV (XPSGN-1)
AAD4  FDB  XPABS-1
AAD4 +B0AD  DW  REV (XPABS-1)
AAD6  FDB  XPINT-1
AAD6 +B0DC  DW  REV (XPINT-1)
AAD8  FDB  XPPLDL-1
AAD8 +B021  DW  REV (XPPLDL-1)
AADA  FDB  XPSTICK-1
AADA +B025  DW  REV (XPSTICK-1)
AADC  FDB  XPPTTRIG-1
AADC +B029  DW  REV (XPPTTRIG-1)
AADE  FDB  XPSTRIG-1
AADE +B02D  DW  REV (XPSTRIG-1)

Execute Expression

AAEØ  LOCAL

EXEXPR — Execute Expression

AAEØ  XLET
AAEØ  EXEXPR
AAEØ  202EAB  JSR  EXPINT  ; GO INIT
AAE3  :EXNXT
AAE3  203EAB  JSR  :EGTOKEN  ; GO GET TOKEN
AAE6  006 "AAE  BCS  :EXOT  ; BR IF OPERATOR
AAE8  20BAAB  JSR  ARG_PUSH  ; PUSH ARG
AAEB  4CE3AA  JMP  :EXNXT  ; GO FOR NEXT TOKEN
AAEE  85AB  :EXOT  STA  EXSVOP  ; SAVE OPERATOR
AAF0  202FAC  TAX
AAF1  BD2FAC  LDA  ORPTAB-16,X  ; GET OP PREC
AAF4  LSRA  ;SHIFT FOR GOES ON TO PREC
AAF4 +A4  LSR  A
AAF5  LSRA  LSR  A
AAF6  LSRA  LSR  A
AAF7  LSRA  LSR  A
AAF7 +A4  LSR  A
AAF8  85AC  STA  EXSVPR  ; SAVE GOES ON PREC
AAFA  85AB  :EXPTST  LDY  OPSTKX  ; GET OP STACK INDEX
AAFC  B180  LDA  [ARGSTK],Y  ; GET TOP OP
AAFE  AA  TAX
AAFF  BD2FAC  LDA  ORPTAB-16,X  ; GET TOP OP PREC
AB02  200F  AND  #$0F
AB04  C5AC  CMP  EXSVPR  ; [TOP OP]: [NEW OP]
AB06  9000 "AB15  BCC  :EPUSH  ; IF T<N, PUSH NEW
        ELSE POP
        AB08  AA  TAX
        AB09  F014 "AB1F  BEQ  :EXEND  ; IF POP SOE
        THEN DONE
        AB0B  EXOPPOP
AB0B  B180  LDA  [ARGSTK],Y  ; RE-GET TOS OP
AB0D  6EA9  INC  OPSTKX
AB0F  282EAB  JSR  :EXOP  ; DEC OF STACK INDEX
AB12  4CFAAA  JMP  :EXPTST  ; GET EXECUTE OP
        ; GO TEST OP WITH NEW TOS
        ; AB15  A5AB  :EPUSH  LDA  EXSVOP  ; GET OP TO PUSH
        ; AB17  80  DEY
        ; AB18  9180  STA  [ARGSTK],Y  ; SET OP IN STACK
        ; AB1A  84A9  STY  OPSTKX
        ; AB1C  4CE3AA  JMP  :EXNXT  ; GO GET NEXT TOKEN
        ; AB1F  XPLPRN

189
Initialize Expression Parameters

GETTOK — Get Next Token and Classify

Source Code
AAPSTR — Pop String Argument and Make Address Absolute

ARGPUSH — Push FR0 to Argument Stack
Source Code

ABCD 88 DEY ; BACKWARDS
ABCE CA DEX
ABCF 10F8 'ABC9 BPL ;APH1 ; DONE
ABD1 60 RTS
ABD2 4C2CB9 ;APERR JMP Errao ; STACK OVERFLOW
GETPINT — Get Positive Integer from Expression
ABD5 GETPINT
ABD5 20E8AB JSR GETINT ; GO GET INT
ABD8 GETPINT
ABD8 A5D5 LDA FR0+1 ; GET HIGH BYTE
ABDA 3001 'ABDD BMI ;GPIERR ; BR > 32767
ABDC 60 RTS ; DONE
ABDD 4C32B9 ;GPIERR JMP ERRLN
GETINT — Get Integer from Expression
ABE0 20E8AA GETINT JSR EXEXPR ; EVAL Expr
ABE2 GETINT
ABE3 28F2AB JSR ARGPOP ; POP VALUE TO FR0
ABE6 4C56AD JMP CVFPI ; GO CONVERT FR0 TO INT & RETURN
GET1INT — Get One-Byte Integer from Expression
ABE9 GET1INT
ABE9 20D5AB JSR GETPINT ; GET INT <32768
ABEC D001 'ABEF BNE ;ERV1 ; IF NOT 1 BYTE, THEN ERROR
ABEE 60 RTS
ABEF :ERV1
ABEF 203AB9 JSR ERVAL
ARGPOP — Pop Argument Stack Entry to FR0 or FR1
ABF2 ARGPOP
ABF2 A5AA LDA ARSLVL ; GET ARG STACK LEVEL
ABF4 C6AA DEC ARSLVL ; DEC AS LEVEL
ABF6 ASLA
ABF6 +0A ASL A
ABF7 ASLA
ABF7 +0A ASL A
ABF8 ASLA
ABF8 +0A ASL A
ABF9 A8 TAY ; Y = START OF NEXT ENTRY
ABFA 88 DEY ; MINUS ONE
ABFB A287 LDX #7 ; X = 7 FOR 8
; AFBF B180 :APOP0 LDA [ARGOPS],Y ; MOVE ARG ENTRY
ABFF 95D2 STA VTYPE,X ; BACKWARDS
AC01 88 DEY
AC02 CA DEX
AC03 10F8 'ABFD BPL ;APOP0
AC05 60 RTS ; DONE
ARGP2 — Pop TOS to FR1,TOS-1 to FR0
AC06 20F2AB ARGP2 JSR ARGPOP ; POP TOS TO FR0
AC09 20B6DD JSR MV$TO1 ; MOV$ FR0 TO FR1
AC0C 4CF2AB JMP ARGPOP ; POP TOS TO FR0 AND RETURN
POP1 — Get a Value in FR0
; ; - EVALUATE EXPRESSION IN STMT LINE & POP IT INTO FR0
; ;
AC0F POP1
AC0F 20E8AA JSR EXEXPR ; EVALUATE EXPRESSION
AC12 20F2AB JSR ARGPOP ; PUSH INTO FR0
AC15 60 RTS
Source Code

RTNVAR — Return Variable to Variable Value Table from FR0

AC16 RTNVAR
AC16 A5D3 LDA VNUM ; GET VAR NUMBER
AC18 2028C JSR GVVTADR
AC1B A200 LDX #0
AC1D B5D2 ;RV1 LDA VTYPE,X ; MOVE FR0 TO
AC1F 919D STA [WVVTPT],X ; VAR VALUE TABLE
AC21 C8 INY
AC22 E8 INX
AC23 E008 CPX #8
AC25 90F6 ACID BCC #RV1 ; DONE
AC27 60 RTS

GVVTADR — Get Value's Value Table Entry Address

AC28 GVVTADR
AC28 A000 LDY #0 ; CLEAR ADR HI
AC2A 849E STY WVVTPT+1 ; MULT VAR NO
AC2C AC1A ASL A ; BY 8
AC2E AC1 A ASL A ; THEN
AC30 ACIA ASL A ; FORM ENTRY
AC32 65B6 ADC VVT ; ADD VVTPT VALUE
AC34 859D STA WVVTPT ; ADR
AC36 A5B7 LDA VVTPT+1
AC38 659E ADC WVVTPT+1
AC3A 859E STA WVVTPT+1
AC3C 60 RTS

Operator Precedence Table

; - ENTRIES MUST BE IN SAME ORDER AS OPRTAB
; - LEFT NIBBLE IS TO GO ON STACK PREC
; - RIGHT NIBBLE IS COME OFF STACK PREC

AC3F 00 DB 00 ; CDQ
AC40 00 DB 00 ; CSOE
AC41 00 DB 00 ; CCOM
AC42 00 DB 00 ; CDOL
AC43 00 DB 00 ; CEOS
AC44 00 DB 00 ; CSE
AC45 00 DB 00 ; CCR
AC46 00 DB 00 ; CTO
AC47 00 DB 00 ; CTO
AC48 00 DB 00 ; CSTEP
AC49 00 DB 00 ; CTHEN
AC4A 00 DB 00 ; CPND
AC4B 00 DB 00 ; CLE
AC4C 88 DB 88 ; CLE
AC4D 88 DB 88 ; CNE
AC4E 88 DB 88 ; CGE
AC4F 88 DB 88 ; CGT
AC50 88 DB 88 ; CLT
AC51 88 DB 88 ; CEQ
AC52 CC DB $CC ; CEXP
AC53 AA DB $AA ; CMUL
AC54 99 DB $99 ; CPLUS
AC55 99 DB $99 ; CMINUS
AC56 AA DB $AA ; CDIV
AC57 77 DB $77 ; CNOT
AC58 55 DB $55 ; COR
AC59 66 DB $66 ; CAND
AC5A F2 DB $F2 ; CLPRN
Source Code

```
AC5B 4E DB  $4E ; CRPRN
AC5C F1 DB  $F1 ; CAASN
AC5D F1 DB  $F1 ; CSASN
AC5E EE DB  $EE ; CSLE
AC5F EE DB  $EE ; CSNE
AC60 EE DB  $EE ; CSGE
AC61 EE DB  $EE ; CSGT
AC62 EE DB  $EE ; CSEQ
AC63 EE DB  $EE ; CUPlus
AC64 DD DB  $DD ; CUMINUS
AC65 DD DB  $DD ; CSLPRN
AC66 F2 DB  $F2 ; CALPRN
AC67 F2 DB  $F2 ; CDLPRN
AC68 F2 DB  $F2 ; CFLPRN
AC69 F2 DB  $F2 ; CDSLPR
AC6A F2 DB  $F2 ; CACOM
AC6B 43 DB  $43
AC6C F2 DB  $F2 ; FUNCTIONS
AC6D F2 DB  $F2
AC6E F2 DB  $F2
AC6F F2 DB  $F2
AC70 F2 DB  $F2
AC71 F2 DB  $F2
AC72 F2 DB  $F2
AC73 F2 DB  $F2
AC74 F2 DB  $F2
AC75 F2 DB  $F2
AC76 F2 DB  $F2
AC77 F2 DB  $F2
AC78 F2 DB  $F2
AC79 F2 DB  $F2
AC7A F2 DB  $F2
AC7B F2 DB  $F2
AC7C F2 DB  $F2
AC7D F2 DB  $F2
AC7E F2 DB  $F2
AC7F F2 DB  $F2
AC80 F2 DB  $F2
AC81 F2 DB  $F2
AC82 F2 DB  $F2
AC83 F2 DB  $F2

Miscellaneous Operators

Miscellaneous Operators' Executors

AC84 XPPLUS JSR ARGEP2
AC87 283BAD JSR FRADD
AC8A 4CBAAB JMP ARGEPUSH
AC8B XPMINUS JSR ARGEP2
AC8D 2806AC JSR PRSUB
AC8E 2847AD JSR FRMUL
AC8F 4CBAAB JMP ARGEPUSH
AC96 XPDIV JSR ARGEP2
AC97 2806AC JSR FRDIV
AC98 4CBAAB JMP ARGEPUSH
AC99 XPUMINUS JSR ARGEP2 ; GET ARGUMENT INTO FR0
AC9A 2806AC JSR ARGEP2 ; GET BYTE WITH SIGN
AC9B 2804AD JSR FRDIV ; FLIP SIGN BIT
AC9C 4CBAAB JMP ARGEPUSH
AC9D XPUMINUS JSR ARGEP2 ; RETURN BYTE WITH SIGN CHANGED
AC9E 4CBAAB JMP ARGEPUSH ; PUSH ON STACKS
AC9F XPPLUS
ACB4 4CBAAB JMP ARGEPUSH
```
Source Code

ACB4 60 RTS
ACB5 XPLE
ACB5 XPSLE
ACB5 2026AD JSR XCMP
ACB6 304D AD05 BMI XTRUE
ACBA F049 AD05 BEQ XTRUE
ACBC 1042 AD00 BPL XFALSE
ACBE XPNE
ACBE XPSNE
ACBE 2026AD JSR XCMP
ACCC 303B AD05 BMI XFALSE
ACCA 1034 AD00 BPL XFALSE
ACCB XPNE
ACCB XPSNE
ACCB 2026AD JSR XCMP
ACCD 3026 AD00 BMI XFALSE
ACCD 1029 AD05 BPL XTRUE
ACCE XPGE
ACCE XPSGE
ACCE 2026AD JSR XCMP
ACCE 1029 AD05 BPL XTRUE
ACCE XPLT
ACCE XPSLT
ACCE 2026AD JSR XCMP
ACCE 08 AD05 BEQ XFALSE
ACCA 00 AD00 BMI XFALSE
ACCD 00 AD05 BPL XTRUE
ACCF 00 AD00 BMI XTRUE
ACCF 00 AD05 BPL XFALSE
ACCF XPGE
ACCF XPSGE
ACCF 2026AD JSR XCMP
ACCF 08 AD05 BEQ XTRUE
ACE1 D01D AD00 BNE XFALSE
;
ACCE 006AC JSR ARG2
ACE6 A5D4 LDA FR0
ACEB 25E0 AND FR1
ACEA F044 AD05 BEQ XFALSE
ACEC D017 AD05 BNE XTRUE
ACEE XPOR
ACEE 006AC JSR ARG2
ACF1 A5D4 LDA FR0
ACF3 05E0 ORA FR1
ACF5 F009 AD00 BEQ XFALSE
ACF7 D00C AD05 BNE XTRUE
ACFG XPNOT
ACFG 20F2AB JSR ARGPOP
ACFC A5D4 LDA FR0
ACFE F005 AD05 BEQ XTRUE
;
AD00 XFALSE
AD00 A900 LDA #0
AD02 AB TAY
AD03 F004 AD09 BEQ XTF
;
AD05 XTRUE
AD05 A940 LDA #$40
AD07 XTJ LDY #1
;
AD09 XTF
AD09 B5D4 STA FR0
AD0B 04D5 STY FR0+1
AD0D A2D6 LDX #FR0+2
AD0F A004 LDY #FPPREC-2
AD11 2048DA JSR XXLY
AD14 05D2 STA VTYPE
AD16 XPUSH
AD16 4CBAAB JMP ARGPOP
Source Code

XPSGN — Sign Function

AD19  XPSGN
AD19  20F2AB JSR ARGPOP
AD19  A5D4 LDA FR0
AD1E  F0F6 "AD16 BEQ XPUSH
AD20  10E3 "AD05 BPL XTRUE
AD22  A9C0 LDA #$5C0 ; GET MINUS EXPONENT
AD24  30E1 "AD07 BMI XTI

XCMP — Compare Executor

AD26  XCMP
AD26  A4A9 LDY OPSTKX ; GET OPERATOR THAT
AD28  88 DEY ; GOT US HERE
AD29  B180 LDA [ARGSTK],Y
AD2B  C92F CMP #$CSLE ; IF OP WAS ARITHMETIC
AD2D  9003 "AD32 BCC FRCMPP ; THEN DO FP REG COMP
AD2F  4C81AF JMP STRCMP ; ELSE DO STRING COMPARE
AD32  2066AC FRCMPP JSR ARGP2

FRCMP — Compare Two Floating Point Numbers

* ON ENTRY FR0 & FR1 CONTAIN FLOATING POINT #'s
* * ON EXIT CC = + FR0 > FR1
* * CC = - FR0 < FR1
* * CC = 0 FR0 = FR1

AD35  FRCMP
AD35  2041AD JSR FRSUB ; SUBTRACT FR1 FROM FR0
AD3B  A5D4 LDA FR0 ; GET FR0 EXPONENT
AD3A  60 RTS ; RETURN WITH CC SET

FRADD — Floating Point Add

* DOES NOT RETURN IF ERROR
* *
AD3B  FRADD
AD3B  2066DA JSR FADD ; ADD TWO #
AD3E  B013 "AD53 BCS :ERROV ; BR IF ERROR
AD40  60 RTS

FRSUB — Floating Point Subtract

* DOES NOT RETURN IF ERROR
* *
AD41  FRSUB
AD41  2066DA JSR FSUB ; SUB TWO #
AD44  B00D "AD53 BCS :ERROV ; BR IF ERROR
AD46  60 RTS

FRMUL — Floating Point Multiply

* DOES NOT RETURN IF ERROR
* *
AD47  FRMUL
AD47  20DBDA JSR FMUL ; MULT TWO #
AD4A  B007 "AD53 BCS :ERROV ; BR IF ERROR
AD4C  60 RTS

FRDIV — Floating Point Divide

* DOES NOT RETURN IF ERROR
* *
AD4D  FRDIV
AD4D  2028DB JSR FDIV ; DIVIDE TWO #
Source Code

AD50 B001 'AD53 BCS :EROV ; BR IF ERROR
AD52 60 RTS ;
; ; AD53 :EROV
AD53 202AB9 JSR ERORV

CVFPI — Convert Floating Point to Integer

* DOES NOT RETURN IF ERROR
*
AD56 CVFPI
AD56 2002D9 JSR FPI ; GO CONVERT TO INTEGER
AD59 B001 'AD5C BCS :ERRVAL ; IF ERROR, BR
AD5B 60 RTS ; ELSE RETURN
; ; AD5C :ERRVAL
AD5C 203AB9 JSR ERVAL ; VALUE ERROR

XPAASN — Arithmetic Assignment Operator

AD5F XPAASN
AD5F A5A9 LDA OPSTKX ; GET OP STACK INDEX
AD61 C9FF CMP #$FF ; AT STACK START
AD63 D00F 'AD74 BNE :AAMAT ; BR IF NOT, [MAT ASSIGN]
; DO SCALER ASSIGN
AD65 2006AC JSR ARGP2 ; GO POP TOP 2 ARGS
AD6B A285 LDX #$5 ; MOVE FR1 VALUE
AD6A B580 'AASN1 LDA FR1,X ; TO FR0
AD6C 95D4 STA FR0,X
AD6E CA DEX
AD6F 10F9 'AD6A BPL :AASN1 ; FR0 TO VVT & RETURN
AD71 4C16AC JMP RTNVAR
; AAMAT
AD74 'AASN1 LDA #$80 ; SET ASSIGN FLAG BIT ON
AD76 85B1 STA ADFLAG ; IN ASSIGN/DIM FLAG
AD78 60 RTS ; GO POP REM OFF OPS

XPACOM — Array Comma Operator

AD79 XPACOM
AD79 E6B0 INC COMCNT ; INCREMENT COMMA COUNT

XPRPRN — Right Parenthesis Operator

; XPFLPRN — FUNCTION RIGHT PAREN OPERATOR
;
AD7B XPRPRN
AD7B XPFLPRN
AD7B A4A9 LDY OPSTKX ; GET OPERATOR STACK TOP
AD7D 68 PLA
AD7E 68 PLA
AD7F 4C0BAB JMP EXOPOP ; GO POP AND EXECUTE NEXT OPERATOR

XPDLPRN — DIM Left Parenthesis Operator

AD82 XDPDSL<br>AD82 XPDLPRN<br>AD82 A940 LDA #$540 ; SET DIM FLAG BIT<br>AD84 85B1 STA ADFLAG ; IN ADFLAG FALL THRU TO XPALPRN

197
XPALPRN — Array Left Parenthesis Operator

AD86 24B1 XPALPRN ; IF NOT ASSIGN
AD86 1006 "AD90 BIT ADFLAG ; THE BRANCH
AD86 1006 "AD90 BPL :ALP1 ELSE
AD8A A5AA LDA ARSLVL ; SAVE STACK LEVEL
AD8C 85AF STA ATEMP ; OF THE VALUE ASSIGNMENT
AD8E 6AA DEC ARSLVL ; AND PSEUDO POP IT
;
AD90 A900 :ALP1 LDA #0 ; INIT FOR I2 = 0
AD92 A8 TAY
AD93 C5B0 CMP COMCNT ; IF COMMA COUNT =0 THEN
AD95 F000 "AD92 BEQ :ALP2
;
AD97 C600 DEC COMCNT ; BR WITH I2 = 0 ELSE
AD99 20E3AB JSR GTINTO ; ELSE POP I2 AND MAKE INT
AD9C A5D5 LDA FR0+1
AD9E 3012 "ADC3 BMI :ALPER ; ERROR IF > 32,767
AD9A A4D4 LDY FR0
;
ADA2 8598 :ALP2 STA INDEX2+1 ; SET 12 VALUE
ADA4 8497 STY INDEX2
;
ADA6 20E3AB JSR GTINTO ; POP I2 AND MAKE INT
ADAB 85F5 LDA ZTEMP1 ; TO ZTEMP1
ADAD A5D5 LDA FR0+1
ADAF 3012 "ADC3 BMI :ALPER ; ERROR IF > 32,767
ADB1 85F6 STA ZTEMP1+1
;
ADB3 20F2AB JSR ARGPOP ; POP THE ARRAY ENTRY
ADB6 24B1 BIT ADFLAG ; IF NOT EXECUTING DIM
ADBO 5005 "ADBF BVC :ALP3 ; THEN CONTINUE
ADBA A900 LDA #0
ADBC 85B1 STA ADFLAG ; IN ADFLAG
ADBE 60 RTS
;
ADBF :ALP3
ADBF 66D2 ROR VTYPE ; IF ARRAY HAS BEEN
ADC1 B003 "ADC6 BCS :ALP4 ; DIMED THEN CONTINUE
ADC3 202EB9 :ALPER JSR ERRDIM ; ELSE DIM ERROR
;
ADC6 :ALP4
ADC6 A5F6 LDA ZTEMP1+1 ; TEST INDEX 1
ADCD C5D7 CMP VTYPE+EVAD1+1 ; IN RANGE WITH
ADCA 9008 "ADD4 BCC :ALP5 ; DIM1
ADCC D0F5 "ADC3 BNE :ALPER
ADCE A5F5 LDA ZTEMP1
ADDB 85B1 CMP VTYPE+EVAD1
ADDE B000 "ADC3 BCS :ALPER
;
ADD4 A598 :ALP3 LDA INDEX2+1 ; TEST INDEX 2
ADD6 C599 CMP VTYPE+EVAD2+1 ; IN RANGE WITH
ADD8 9008 "ADE2 BCC :ALP6 ; DIM 2
ADDA D087 "ADC3 BNE :ALPER
ADDC A597 LDA INDEX2
ADDE C5D8 CMP VTYPE+EVAD2
ADE0 B0E1 "ADC3 BCS :ALPER
;
ADE2 205DAF :ALP6 JSR AMUL1 ; INDEX1 = INDEX1
ADE5 A597 LDA INDEX2 ; INDEX1 = INDEX1 + INDEX2
ADE7 A498 LDY INDEX2+1
ADE9 2052AF JSR AADD
ADEC 2046AF JSR AMUL2 ; ZTEMP1 = ZTEMP1*6
ADEP A5D4 LDA VTYPE+EVAD1 ; ZTEMP1 = ZTEMP1 + DISPL
ADF1 A4D5 LDY VTYPE+EVAD1+1
ADF3 2052AF JSR AADD
ADF6 A5BC LDA STARP ; ZTEMP1 = ZTEMP1 + ADR
ADF8 A4BD LDY STARP+1
ADF9 2052AF JSR AADD ZTEMP1 NOW POINTS
; TO ELEMENT REQD
; IF NOT ASSIGN ELSE ASSIGN
ADF0 2481 BIT ADFLAG ; THEN CONTINUE
ADF1 1015 A816 BPL :ALP8
ADF2 A5AF LDA ATEMP ; RESTORE ARG LEVEL
ADF3 85AA STA ARSLVL ; TO VALUE AND
ADF4 20F2AB JSR ARGPOP ; POP VALUE
;
ADF5 A005 LDY #5
ADF6 A9D400 :ALP7 LDA FR0,Y ; MOVE VALUE
ADF7 91F5 STA [ZTEMP1],Y ; TO ELEMENT SPACE
ADF8 8B DEY
ADF9 18F8 A81A BPL :ALP7
ADF10 C8 INY ; TURN OFF
ADF11 84B1 STY ADFLAG ; ADFLAG
ADF12 68 RTS ; DONE
;
ADF13 A005 :ALP8 LDY #5
ADF14 B1F5 :ALP9 LDA [ZTEMP1],Y ; MOVE ELEMENT TO
ADF15 99D400 STA FR0,Y ; FR0
ADF16 8B DEY
ADF17 18F8 A818 BPL :ALP9
ADF18 C8 INY
ADF19 84D2 STY VTYPE
ADF20 4CBAAAB JMP ARGPUSH ; PUSH FR0 BACK TO STACK AND RETURN

XPSLPRN — String Left Parenthesis

AE26 XPSLPRN
AE26 A5B0 LDA COMCNT ; IF NO INDEX 2
AE27 F807 A831 BEQ :XSLP2 ; THEN BR
;
AE28 2096AE JSR :XSPV ; ELSE POP 12 AND
AE29 8498 STY INDEX2+1 ; SAVE IN INDEX 2
AE2A 8597 STA INDEX2
;
AE31 2096AE :XSLP2 JSR :XSPV ; POP INDEX 1
AE32 38 SEC ; ADD DECREMENT BY ONE
AE33 E901 SBC #1 ; AND PUT INTO ZTEMP1
AE34 85F5 STA ZTEMP1
AE35 9B TYA
AE36 E900 SBC #0
AE37 85F6 STA ZTEMP1+1
;
AE38 20F2AB JSR ARGPUSH ; POP ARG STRING
;
AE41 A5B1 LDA ADFLAG ; IF NOT A DEST STRING
AE42 10B0 A850 BPL :XSLP3 ; THEN BRANCH
AE43 05B0 ORA COMCNT
AE44 85B1 STA ADFLAG
AE45 A4D9 LDY VTYPE+EVSDIM+1 ; INDEX 2 LIMIT
AE46 A5D8 LDA VTYPE+EVSDIM ; IS DIM
AE47 4C54AE JMP :XSLP4
;
AE50 A5D6 :XSLP3 LDA VTYPE+EVSLLEN ; INDEX 2 LIMIT
AE51 A4D7 LDY VTYPE+EVSLLEN+1 ; IS STRING LENGTH
;
AE54 A6B0 :XSLP4 LDX COMCNT ; IF NO INDEX 2
AE55 F810 A868 BEQ :XSLP6 ; THEN BRANCH
AE56 C6B0 DEC COMCNT ; ELSE
AE57 A498 CPY INDEX2+1
AE58 9035 A893 BCC :XSLER
AE59 D804 A864 BNE :XSLP5 ; INDEX 2 LIMIT
AE60 C597 CMP INDEX2
AE61 902F A893 BCC :XSLER

199
Source Code

```
AE64 A498  :XSLP5  LDY  INDEX2+1 ;USE INDEX 2
AE66 A597   LDA  INDEX2 ;AS LIMIT ;
AE68 38     :XSLP6  SEC ; LENGTH IS
AE69 E5F5   SBC  ZTEMP1
AE6B 85D6   STA  VTYPE+EVSLEN ; LIMIT - INDEX 1
AE6D AA     TAX
AE6E 98     TYA
AE6F E5F6   SBC  ZTEMP1+1
AE71 85D7   STA  VTYPE+EVSLEN+1
AE73 9B1E  *AE93  BCC  :XSLER ; LENGTH MUST BE
AE75 A8     TAY ; GE ZERO
AE76 D003  *AE7B  BNE  :XSLP7
AE78 8A     TXA
AE79 F018  *AE93  BEQ  :XSLER ;
AE7B 209B4B  :XSLP7  JSR  GSTRAD ; GET ABS ADR ;
AE7E 18     CLC
AE7F A5D4   LDA  VTYPE+EVSADR
AE81 65F5   ADC  ZTEMP1 ; STRING ADR
AE83 85D4   STA  VTYPE+EVSADR ; STRING ADR + INDEX 1
AE85 A5D5   LDA  VTYPE+EVSADR+1
AE87 65F6   ADC  ZTEMP1+1
AE89 85D5   STA  VTYPE+EVSADR+1 ;
AE8B 24B1   BIT  ADFLAG ; IF NOT ASSIGN
AE8D 1301  *AE90  BPL  :XSLP8 ; THEN BR ;
AE8F 68     RTS ; ELSE RETURN TO ASSIGN ;
AE90 4CBAAB  :XSLP8  JMP  ARGPUH ; PUSH ARG AND RETURN ;
AE93 2036B9  :XSLER  JSR  ERRSSL

XSPV — Pop Index Value as Integer and Insure Not Zero

AE96  :XSPV
AE98 A5D4   LDA  VTYPE+EVSADR
AE9A 85D4   STA  VTYPE+EVSADR
AE9C 859A   STA  MVFA+1
AE9E A5D6   LDA  VTYPE+EVSLEN ; MVFLG = LENGTH
AE98 B5A2   STA  MVFLG+1
AE9A A4D7   LDY  VTYPE+EVSLEN+1
AE9C 84A3   STY  MVFLG+1 ;
AE9E A4A9   LDY  OPSTKX ; IF AT TOP OF
AE98 C0FF   CPY  #$FF ; OP STACK
AE9A F00F  *AECB  BEQ  :XSA1 ; THEN BR ;
AE9C A980   LDA  #$80 ; SET ASSIGN BIT
AE9E 85B1   STA  ADFLAG ; IN ASSIGN/DIM FLAG
AEAC 2800AB  JSR  EXOPPOP ; AND PROCESS SUBSTRING
AEAD A5D7   LDA  VTYPE+EVSLEN+1 ; A, Y =
AEAE A4D6   LDY  VTYPE+EVSLEN ; DEST LEN
AEAC 26B1   ROL  ADFLAG ; TURN OFF ASSIGN
AEAC B007  *AED2  BCS  :XSA2A ; AND BR

XSAASN — String Assign Operator

A8A3  XSAASN
A8A4 209BA8  JSR  AAPSTR ; POP STR WITH ABS ADR
A8A6  RISAASN
A8A6 A5D4   LDA  VTYPE+EVSADR ; MVFA = ADR
A8A8 8599   STA  MVFA
A8A9 A5D5   LDA  VTYPE+EVSADR+1
A8AA 859A   STA  MVFA+1
A8AB A5D6   LDA  VTYPE+EVSLEN
A8AC B5A2   STA  MVFLG
A8AE A4D7   LDY  VTYPE+EVSLEN+1
A8AC 84A3   STY  MVFLG+1 ;
A8A6 A4A9   LDY  OPSTKX ; IF AT TOP OF
A8A8 C0FF   CPY  #$FF ; OP STACK
A8A9 F00F  *AECB  BEQ  :XSA1 ; THEN BR ;
A8AC A980   LDA  #$80 ; SET ASSIGN BIT
A8AE 85B1   STA  ADFLAG ; IN ASSIGN/DIM FLAG
A8AD 2800AB  JSR  EXOPPOP ; AND PROCESS SUBSTRING
A8AE A5D7   LDA  VTYPE+EVSLEN+1 ; A, Y =
A8BF A4D6   LDY  VTYPE+EVSLEN
A8AC 26B1   ROL  ADFLAG ; TURN OFF ASSIGN
A8AC B007  *AED2  BCS  :XSA2A ; AND BR

200
```
Source Code

AECB 2098AB :XSA1 JSR AAPSTR ; POP STR WITH ABS ADR

AEC0 A5D9 :XSA2 LDA VTYPE+EVS+DIM+1 ; A, Y = DEST LENGTH

AED0 A4D8 LDY VTYPE+EVS+DIM

AED2 C5A3 CMP MVLNG+1 ; IF DEST LENGTH

AED4 9006 ^AEDC BCC :XSA3 ; LESS THAN MOVE LENGTH

AED6 D008 ^AEE0 BNE :XSA4

AED9 C4A2 CPY MVLNG ; THEN

AEDB 85A3 :XSA3 STA MVLNG+1 ; SET MOVE LENGTH

AEDD 84A2 STY MVLNG ; = DIST LENGTH

AEE0 18 :XSA4 CLC

AEE1 A5D4 LDA VTYPE+EVADDR ; MOVE LENGTH PLUS

AEE3 65A2 ADC MVLNG ; START ADDR IS

AEE5 A6 TAX

AEE6 A5D5 LDA VTYPE+EVADDR+1

AEE8 65A3 ADC MVLNG+1

AEEA AA TAX

AEEB 38 SEC ; END ADDR MINUS

AEEC 9B TYA ; START OF STRING

AED0 E5A0 SBC STARP ; SPACE IS DISPL

AEEF 85F9 STA ZTEMP3 ; TO END OF STRING

AEE1 8A TXA ; WHICH WE SAVE

AEE5 85D0 SBC STARP+1 ; IN ZTEMP3

AEE6 85FA STA ZTEMP3+1

AEE6 38 SEC ; SET MOVE LENGTH LOW

AEE7 A900 LDA #$0 ; = $100 - MVL [L]

AEE9 E5A2 SBC MVLNG ; BECAUSE OF THE WAY

AEEB 85A2 STA MVLNG ; FMOVE WORKS

AEEF 38 SEC

AEEF A599 LDA MVFA ; ADJUST MVFA TO

AEFF 85A2 SBC MVLNG ; CONFORM WITH MVFA

AEFG 8598 STA MVFA ; CHANGE

AEF0 A59A LDA MVFA+1

AEF5 E900 SBC #$0 ; WITH MVL

AEF8 859A STA MVFA+1

AEF0 A5D4 LDA VTYPE+EVADDR ; MOVE THE DEST

AEF9 85D6 SBC MVLNG ; STRING ADDR TO

AEFD A90B STA MVTB ; MVTB AND

AEFF A5D5 LDA VTYPE+EVADDR+1 ; MAKE IT CONFORM

AEF1 E900 SBC #$0 ; WITH MVL

AEF4 859C STA MVTB+1

AEF7 2047A9 JSR FMOVER ; GO DO THE VERY FAST MOVE

AEF9 A5D3 LDA VNUN ; GO GET ORIGNAL DEST

AEFB 2089AB JSR GETVAR ; STRING

AEF6 38 SEC

AEF8 A599 LDA MVFA ; ADJUST MVFA TO

AEFF 85A2 SBC MVLNG ; CONFORM WITH MVFA

AEFG 8598 STA MVFA ; CHANGE

AEF0 A59A LDA MVFA+1

AF00 A5D4 LDA VTYPE+EVADDR ; MOVE THE DEST

AF00 85D6 SBC MVLNG ; STRING ADDR TO

AF05 859B STA MVTB ; MVTB AND

AF09 A5D5 LDA VTYPE+EVADDR+1 ; MAKE IT CONFORM

AF13 E900 SBC #$0 ; WITH MVL

AF16 859C STA MVTB+1

AF17 2047A9 JSR FMOVER ; GO DO THE VERY FAST MOVE

AF19 A5D3 LDA VNUN ; GO GET ORIGNAL DEST

AF1C 2089AB JSR GETVAR ; STRING

AF1F 38 SEC

AF20 A5F9 LDA ZTEMP3 ; DISPL TO END OF

AF22 85D6 SBC VTYPE+EVADDR ; MOVE MINUS DISPL

AF24 A8 TAY ; TO START OF STRING

AF25 A5FA LDA ZTEMP3+1 ; IS OUR RESULT LENGTH

AF27 85D5 SBC VTYPE+EVADDR+1

AF29 AA TAX

AF2A A902 LDA #$0 ; IF THE DESTINATION

AF2C 2581 AND ADFLAG ; LENGTH WAS IMPLICIT

AF2E E90F ^AF3F BEQ :XSA5 ; SET NEW LENGTH

AF30 A900 LDA #$0 ; CLEAR

201
Source Code

AMUL2 — Integer Multiplication of ZTEMP1 by 6

AADD — Integer Addition of [A,Y] to ZTEMP1

AMUL — Integer Multiplication of ZTEMP1 by DIM2

STRCMP — String Compare
Source Code

AF8A A2D6 :SCI LDX #$F0-2+EVSLEN ; GO DEC STR A LEN
AF8C 20BCAF JSR ZPADEC
AF89 08 PHP ; SAVE RTN CODE
AF90 A2E2 LDX #$F1-2+EVSLEN ; GO DEC STR B LEN
AF95 F013 'AFAA BEQ :SCLT ; BR STR B LEN = 0
AF97 28 PLP ; GET STR A COND CODE
AF98 F000D 'AF7 BEQ :SCLT ; BR STR A LEN = 0

AF9A A000 LDY #0 ; COMPARE A BYTE
AF9C B1D4 LDA [FR0-2+EVSADR],Y ; OF STRING A
AF9E D1E0 CMP [FR1-2+EVSADR],Y ; TO STRING B
AF90 F00C 'AFAE BEQ :SC3 ; BR IF SAME
AF92 9003 'AF7 BCC :SCLT ; BR IF A<B

AF94 A091 :SCGT LDA #1 ; A>B
AF96 60 RTS

AF89 28 :SC2 PLP ; IF STR A LEN NOT
AF8B D0F7 'AF4 BNE :SCGT ; ZERO THEN A>B
AF8D 60 :SCEQ RTS ; ELSE A=B
AF8E E6D4 'AF6 INE FR0-2+EVSADR ; INC STR A ADR
AF8B D0F2 'AF4 BNE :SC4 ; INC FR0-2+EVSADR
AF82 E6D5 INC FR1-2+EVSADR ; INC STR B ADR
AF86 D0F2 'AF8 BNE :SC1
AF88 E6E1 INC FR1-1+EVSADR
AF89 8CE 'AF8 BNE :SC1

ZPADEC — Decrement a Zero-Page Double Word
AF8C ZPADEC
AF8E B500 LDA 0,X ; GET LOW BYSTE
AF88 D0F6 'AF6 BNE :ZPAD1 ; BR NOT ZERO
AF83 B501 LDA 1,X ; GET HI BYTE
AF82 F005 'AF9 BEQ :ZPADR ; BR IF ZERO
AF84 E6D1 DEC 1,X ; DEC HIGH BYTE
AF86 D609 :ZPAD1 DEC 0,X ; SET NE COND CODE
AF88 A8 TAY ; RETURN
AF89 60 :ZPADR RTS

Functions

XPLEN — Length Function

AFCA XPLEN
AFCA 2098AB JSR AAPSTR ; POP STRING WITH ABS ADR
AFCD A5D6 LDA VTYPE+EVSLEN ; MOVE LENGTH
AFCF A4D7 LDY VTYPE+EVSLEN+1
AFD1 XPIFF
AFD1 85D4 STA FR0 ; TO TOP OF FR0
AFD3 84D5 STY FR0+1
AFD5 20AAD9 XPIFF1 JSR CVIFF ; AND CONVERT TO FP
AFD8 XPIFF2

AFD8 A900 LDA #0 ; CLEAR
AFDA 85D2 STA VTYPE ; TYPE AND
AFDC 85D3 STA VNUM ; NUMBER
AFDE 4CBAA8 JMP ARG_PUSH ; PUSH TO STACK AND RETURN

XPPEEK — PEEK Function

AFE1 XPPEEK
AFE1 20E3AB JSR GTINTO ; GET INT ARG
AFE4 A000 LDY #0 ; GET MEM BYTE
AFE6 B1D4 LDA [FR0],Y
AFED 4CD1AF JMP XPIFF ; GO PUSH AS FP
Source Code

XPFR – FRE Function

```
AFEB 20F2AB JSR ARGPOP ; POP DUMMY ARG
AFEB 38 SEC
AFEF ADE502 LDA HIMEM ; NO FREE BYTES
AFF2 E590 SBC MEMTOP ; = HIMEM-MEMTOP
AFF4 8504 STA PR0
AFF6 ADE502 LDA HIMEM+1
AFF9 E591 SBC MEMTOP+1
AFFB 8505 STA PR0+1
AFFD 4CD5AF JMP XPIFP1 ; GO PUSH AS FP
```

XPVAL – VAL Function

```
B000 2079BD JSR SETSEOL ; PUT EOL AT STR END
B003 A900 LDA $0 ; GET NUMERIC TERMINATOR
B005 05F2 STA CIX ; GET INDEX INTO BUFFER = 0
B007 2000D8 JSR CVAPF ; CONVERT TO FP.
```

Restore Character

```
B00A 2099BD JSR RSTSEOL ; RESET END OF STR
B00D 90C9 'AFD8 BCC XPIFP2 ;
B00F 'VERR
B00F 201CB9 JSR ERSVAL
```

XPASC – ASC Function

```
B012 XPASC B012 2098AB JSR AAPSTR ; GET STRING ELEMENT
```

Get 1>T Byte of String

```
B015 A000 LDY #0 ; GET INDEX TO 1ST BYTE
B017 B1D4 LDA [PR0-2+EVSADR],Y ; GET BYTE
B019 4CD1AF JMP XPIFP ;
B01C XPADRU
B01C 2098AB JSR AAPSTR ; GET STRING
B01F 4CD5AF JMP XPIFP1 ; FINISH
```

XPPDL – Function Paddle

```
B022 XPDL B022 A900 LDA $0 ; GET DISPL FROM BASE ADDR
B024 F00A 8B03 BEQ :GRF
```

XPSTICK – Function Joystick

```
B026 XPSTICK B026 A908 LDA $8 ; GET DISP FROM BASE ADDR
B028 D006 8B03 BNE :GRF
```

XPPTTRIG – Function Paddle Trigger

```
B02A XPPTTRIG B02A A90C LDA #$0C ; GET DISPL FROM BASE ADDR
B02C D002 8B03 BNE :GRF
```

XPSTRIG – Function Joystick Trigger

```
B02E XPSTRIG B02E A914 LDA #$14 ; GET DISP FROM BASE ADDR
```
Source Code

; Source Code

; GET INTEGER FROM STACK

; SOURCE CODE

; GET VALUE IN FR0

; CONVERT TO ASCII

; SET ADDR

; INIT FOR LENGTH COUNTER

; BUMP COUNT

; GET CHAR

; IS MSB NOT ON, REPEAT

; TURN OFF MSB

; RETURNS CHAR TO BUFFER

; INC TO GET LENGTH

; SET LENGTH LOW

; JOIN CHR FUNCTION

; GET VALUE IN FR0

; CONVERT TO INTEGER

; GET INTEGER LOW

; SET ADDR

; SET LENGTH LOW

; SET LENGTH HIGH
B082 85D3  STA  VNUM ; CLEAR VARIABLE #
B084 A9B3  LDA #EVSTR+EVSSTA+EVDIM ; GET TYPE FLAGS
B086 85D2  STA  VTYPE ; SET VARIABLE TYPE
B088 4CBAA6  JMP  ARGPUSh ; PUSH ON STACK

XPRND — RND Function
B08B  XPRND
B08B A2A8  LDX #RNDIV&255 ; POINT TO 65535
B08D A0B0  LDY #RNDIV/256 ; X
B08F 2098DD  JSR  FLD1R ;MOVE IT TO FRI
B092 20F2AB  ; JSR  ARGPOp ; CLEAR DUMMY ARG
B095 AC0A2D  LDY RNDLOC ; GET 2 BYTE RANDOM #
B098 84D4  STY FR0 ; X
B09A AC8AD2  LDY RNDLOC ; X
B09D 84D5  STY FR0+1 ; X
B09F 204AD9  JSR CVIPP ; CONVERT TO INTEGER
B0A2 204DAD  JSR FRDIV ;DO DIVIDE
B0A5 4CBAA6  JMP  ARGPUSh ; PUT ON STACK

B0A8 4206553600  RNDIV  DB $42,$06,$55,$36,0,0

XPABS — Absolute Value Function
B0AE  XPABS
B0AE 20F2AB  JSR  ARGPOp ;GET ARGUMENT
B0B1  A5D4  LDA  FR0 ;GET BYTE WITH SIGN
B0B3  297F  AND  #$7F ;AND OUT SIGN
B0B5  85D4  STA  FR0 ;SAVE
B0B7  4CBAA6  JMP  ARGPUSh ;PUSH ON STACK

XPUSR — USR Function
B0BA  XPUSR
B0BA 20CBAB  JSR  USR ;PUT RETURN ADDR IN CPU STACK
B0BD  20A9D9  JSR  CVIPP ; CONVERT FR0 TO FP
B0C9  4CBAA6  JMP  ARGPUSh ; PUSH ON STACK

B0C3  :USR
B0C3  A5B0  LDA  COMCYN ;GET COMMA COUNT
B0C5  85C6  STA  ZTEMP2 ;SET AS # OF ARG FOR LOOP CONTROL
B0C7  :USR1
B0C7  20E3AB  JSR  GTINTO ; GET AN INTEGER FROM OP STACK
B0CA  06C6  DEC  ZTEMP2 ;DEC # OF ARGUMENTS
B0CC  3009 'B0D7  BMI  :USR2 ;IF DONE THEM ALL, BRANCH
B0C8  A5D4  LDA  FR0 ;GET ARGUMENT LOW
B0D0  48  PHA  ;PUSH ON STACK
B0D1  A5D5  LDA  FR0+1 ;GET ARGUMENT HIGH
B0D3  48  PHA  ;PUSH ON STACK
B0D4  4CC7B0  JMP  :USR1 ;GET NEXT ARGUMENT
B0D7  :USR2
B0D7  A5B0  LDA  COMCYN ;GET # OF ARGUMENTS
B0DA  6CD400  JMP  [FR0] ;GO TO USER ROUTINE

XPINT — Integer Function
B0DD  XPINT
B0DD  20F2AB  JSR  ARGPOp ; GET NUMBER
B0E0  20E6B0  JSR  XINT ; GET INTEGER
B0E3  4CBAA6  JMP  ARGPUSh ; PUSH ON ARGUMENT STACK
XINT — Take Integer Part of FR0

Source Code

XPSIN — Sine Function

XPCOS — Cosine Function

XPATN — Arc Tangent Function

Transcendental Functions

207
Source Code

XPLOG — LOG Function

B139 XPLOG
B139 20F2AB JSR ARGPOP
B13C 20CDDE JSR LOG
B13F B021 "B162 BCS :TBAD
B141 901C "B15F BCC :TGOOD

XPL10 — LOG Base 10

B143 XPL10
B143 20F2AB JSR ARGPOP
B146 20D1DE JSR LOG10
B149 B017 "B162 BCS :TBAD
B14B 9012 "B15F BCC :TGOOD

XPEXP — EXP Function

B14D XPEXP
B14D 20F2AB JSR ARGPPOP
B150 20C0DD JSR EXP
B153 B00D "B162 BCS :TBAD
B155 9008 "B15F BCC :TGOOD

XPSQR — Square Root Function

B157 XPSQR
B157 20F2AB JSR ARGPPOP
B15A 20E5BE JSR SQR
B15D B003 "B162 BCS :TBAD

XPPower — Exponential Operator [A**B]

B165 XPPower
B165 2086AC JSR ARGP2
B168 A5D4 LDA FR0
B16A D00B "B177 BNE :N0
B16C A5E0 LDA FR1
B16E F004 "B174 BEQ :P0
B170 10ED "B15F BPL :TGOOD
B172 30EE "B162 BMI :TBAD
B174 4C05AD JMP XTRUE
B177 :N0
B177 1030 "B1A9 BPL :SPEVEN
B179 297F AND #$7F
B17B 85D4 STA FR0
B17D A5E0 LDA FR1
B17F 297F AND #$7F
B181 3B SEC
B182 E940 SBC #$40
B184 30DC "B162 BMI :TBAD
B186 A206 LDX #6
B188 C905 CMP #5
B18A 9004 "B190 BCC :SP4
B18C A001 LDY #1
B18E D000 "B190 BNE :SP3
B190 :SP4
B190 85F5 STA ZTEMP1

B198 85F5 STA ZTEMP1 ; SAVE EXP -40
Source Code

B192  38  SEC
B193  A905  LDA  $5
B195  85F5  SBC  ZTEMP1
B197  A8  TAY
B198  86  DEX
B199  A8  Phy
B19A  F006  BIA2  BEQ  :SP2
B19C  B5E0  LDA  FR1,X
B19E  B0C2  B162  BNE  :TBAD
B1A0  F0F6  B198  BEQ  :SP3

B1A2  :SP2
B1A2  A800  LDY  #$80
B1A4  B5E0  LDA  FR1,X
B1A6  4A  LSRA  A
B1A7  B002  B1A8  BCS  :POWR
B1A9  :SPEVEN
B1A9  A000  LDY  $0
B1AB  :POWR
B1A8  98  TYA
B1AC  48  PHA

Save Exponent [from FR1]

B1AD  A205  LDX  #FMPREC
B1AF  :POWR1
B1AF  B5E0  LDA  FR1,X
B1B1  48  PHA
B1B2  CA  DEX
B1B3  10FA  B1AF  BPL  :POWR1
B1B5  20D1DE  JSR  LOG10
B1B8  B0A8  B162  BCS  :TBAD

Pull Exponent into FR1 [from CPU Stack]

B1BA  A200  LDX  $0
B1BC  A005  LDY  #FMPREC
B1BE  :POWR2
B1BE  68  PLA
B1BF  95E0  STA  FR1,X
B1C1  E0  INX
B1C2  88  DEY
B1C3  10F6  B1BE  BPL  :POWR2
B1C5  2047AD  JSR  FMUL
B1CB  20CCDD  JSR  EXP10
B1CB  B009  B1D6  BCS  :EROV
B1CD  60  PLA
B1CE  18F6  B15F  BPL  :TGOOD
B1D0  05D4  ORA  FR0
B1D2  85D4  STA  FR0
B1D4  D089  B15F  BNE  :TGOOD
B1D6  202AB9  JSR  EROVFL
**Source Code**

**Statements**

**XDIM & XCOM — Execute DIM and COMMON Statements**

```assembly
BD9 XDIM
BD9 XCOM
;
BD9 A4A8 :DC1 LDY STINDEX ; IF NOT AT
BD9 C4A7 CPY NXTSTD ; STATEMENT END
BD9 9001 B1E0 RTS ; THEN CONTINUE
BD9 60 :DC2 JSR EXEXPR ; GO SET UP VIA EXECUTE EXPR
BD9 B1E3 A5D2 STY VTYPE ; GET VAR TYPE
BD9 B1E5 ROR ; SHIFT DIM BIT TO CARRY
BD9 +6A ROR A
BD9 B1E6 9003 B1EB BCC :DC3 ; CONTINUE IF NOT YET DIMMED
BD9 B1E8 202EB9 :DCERR JSR ERRDIM ; ELSE ERROR
;
B1EB 38 :DC3 SEC ; TURN ON
B1EC ROLA ; DIM FLAG
B1EC +2A ROL A
BD1D 85D2 STA VTYPE ; AND RESET
B1EF 302F B220 BMI :DCSTR ; AND BR IF STRING
;
B1F1 A4F5 LDY ZTEMP1 ; INC 11 BY 1
B1F3 A6F6 LDX ZTEMP1+1 ; AND SET AS DIM1
B1F5 CB INY
B1F6 D003 B1FB BNE :DC4
B1FB E8 INX
B1F9 30ED B1EB BMI :DCERR ; BR IF OUT OF BOUNDS
BD1F 84D6 :DC4 STY VTYPE+EVAD1
BD1D 86D7 STX VTYPE+EVAD1+1
B1FF 84F5 STY ZTEMP1 ; ALSO PUT BACK ONTO
B281 86F6 STX ZTEMP1+1 ; INDEX 1 FOR MULT
;
B283 A497 LDY INDEX2 ; INC INDEX 2 BY 1
B285 A698 LDX INDEX2+1 ; AND SET AS DIM 2
B287 CB INY
B288 D003 B20D BNE :DC5
B289 E8 INX
B28B 30DB B1EB BMI :DCERR ; BR IF OUT OF BOUNDS
BD2D 84D6 :DC5 STY VTYPE+EVAD2
BD2F 86D9 STX VTYPE+EVAD2+1
;
B211 205DAF JSR AMUL1 ; ZTEMP1 = ZTEMP1*D2
B214 2046AF JSR AMUL2 ; ZTEMP1 = ZTEMP1*6
; RESULT IS AN ARRAY SPACE REQD
;
B217 A4F5 LDY ZTEMP1 ; A,Y = LENGTH
B219 A5F6 LDX ZTEMP1+1
B21B 30CB B1E8 BMI :DCERR
B21D 4C34B2 JMP :DCEXP ; GO EXPAND
;
B220 :DCSTR
B220 A900 LDA #0 ; SET CURRENT LENGTH =0
B222 85D6 STA EVSLEN+VTYPE
B224 85D7 STA EVSLEN+1+VTYPE
;
B226 A4F5 LDY ZTEMP1 ; MOVE INDEX
B228 84D8 STY VTYPE+EVSDIM ; TO STR DIM
B22A A5F6 LDA ZTEMP1+1 ; [ALSO LOAD A,Y]
B22C 85D9 STA VTYPE+EVSDIM+1 ; FOR EXPAND
B22E D004 B234 BNE :DCEXP ; INSURE DIM
B230 C000 CPY #0 ; NOT ZERO
B232 F084 B1E8 BEQ :DCERR ; FOR STRING
;
B234 :DCEXP
B234 A28E LDX #ENDSTAR ; POINT TO END ST & ARRAY SPACE
B236 2081A8 JSR EXPAND ; GO EXPAND
```
Source Code

B239 38 SEC
B23A 5A97 LDA SVESA ; CALCULATE DISPL INTO
B23C 85BC SBC STARP ; ST/ARRAY SPACE
B23E 85D4 STA VTYPE+EVSADR ; AND PUT INTO VALUE BOX
B240 A598 LDA SVESA+1
B242 85BD SBC STARP+1
B244 85D5 STA VTYPE+EVSADR+1
B246 2016AC JSR RTNVAR ; RETURN TO VAR VALUE TABLE
B249 4CD9B1 JMP : DC1 ; AND GO FOR NEXT ONE

XPOKE — Execute POKE

B24C XPOKE
B24C 20E0AB JSR GETINT ; GET INTEGER ADDR
B24F A5D4 LDA FR0 ; SAVE POKE ADDR
B251 8595 STA POKADR ;
B253 A5D5 LDA FR0+1 ;
B255 8596 STA POKADR+1 ;
B257 20E9AB JSR GET1INT ; GET 1 BYTE INTEGER TO POKE
B25A A5D4 LDA FR0 ; GET INTEGER TO POKE
B25C A000 LDY $0 ; GET INDEX
B25E 9195 STA [POKADR],Y ;GET INDEX

XDEG — Execute DEG

B261 XDEG
B261 A906 LDA #DEGON ; GET DEGREES FLAG
B263 85FB STA RADFLG ; SET FOR TRANSCENDENTALS
B265 60 RTS

XRAD — Execute RAD

B266 XRAD
B266 A900 LDA #RADON ; GET RADIUS FLAG
B268 85FB STA RADFLG ; SET FOR TRANSCENDENTALS
B26A 60 RTS

XREST — Execute RESTORE Statement

B26B XREST
B26B A900 LDA $0 ; ZERO DATA DISPL
B26D 85B6 STA DATAD ;
B26F 201B9 JSR TSTEND ; TEST END OF STMT
B272 9003 "B277 BCC :XR1 ; BR IF NOT END
B274 A8 TAY
B275 F007 "B27E BEQ :XR2
B277 20D5AB :XR1 JSR GETPINT ; GET LINE NO.
B27A 85D5 LDA FR0+1 ; LOAD LINE NO.
B27C A4D4 LDY FR0 ;
B27E 85B8 :XR2 STA DATALN+1 ; SET LINE
B280 84B7 STY DATALN
B282 60 RTS ; DONE

XREAD — Execute READ Statement

B283 XREAD
B283 A5A8 LDA STINDEX ; SAVE STINDEX
B285 48 PHA
B286 28C7B6 JSR XGS ; SAVE READ STMT VIA GOSUB
B289 A5B7 LDA DATALN
B28B 85A0 STA TSLNUM
B28D A5B8 LDA DATALN+1
B28F 85A1 STA TSLNUM+1
Source Code

B291 20A2A9 ; JSR GETSTMT ; GO FIND TSLNUM
B294 A5BA ; LDA STMCUR ; MOVE STM CUR TO INBUFF
B296 85F3 ; STA INBUFF
B298 A5BB ; LDA STM CUR+1
B29A 85F4 ; STA INBUFF+1

B29C 2019B7 ; JSR XRTN ; RESTORE READ STMT VIA RETURN
B29F 68 ; PLA
B2A0 85A8 ; STA STINDEX ; SET IT

B2A2 ; :XRD1
B2A2 A000 ; LDY #$0 ; SET CIX=0
B2A4 84F2 ; STY CIX ; SET CIX
B2A6 2007B3 ; JSR :XRNT1 ; GET LINE NO. LOW
B2A9 85B7 ; STA DATALN ; SET LINE NO. LOW
B2AB 2005B3 ; JSR :XRNT ; SET LINE NO. HIGH
B2AE 85B8 ; STA DATALN+1
B2B0 2005B3 ; JSR :XRNT ; SET LINE LENGTH
B2B3 85F5 ; STA ZTEMP1
B2B5 2005B3 ; JSR :XRNT ; SET STM T LENGTH
B2B8 85F6 ; STA ZTEMP1+1

B2C1 84F6 ; LDY ZTEMP1+1 ; GET DISPL TO NEXT STMT
B2C3 84F5 ; CPY ZTEMP1 ; IS IT EOL
B2C5 B005 B2CC ; BCS :XRNTA ; BR IF EOL
B2C7 8B ; DEY
B2C8 84F2 ; STY CIX ; SET NEW DISPL
B2CA 90E9 ; BCC :XRNT ; AND CONTINUE THIS STMT

B2CC 84F2 ; :XRD2A STY CIX
B2CE 84F2 ; DEY

B2D0 A001 ; :XRD3 LDY #$1 ; WAS THIS STMT THE
B2D2 B1F3 ; LDA [INBUFF],Y ; DIRECT ONE
B2D4 3B3D B2B3 ; BMI :XROOD ; BR IF IT WAS [OUT OF DATA]

B2D6 38 ; SEC
B2D7 A5F2 ; LDA CIX ; INBUFF + CIX + 1
B2D9 65F3 ; ADC INBUFF ; = ADR NEXT PGM LINE
B2DB 85F3 ; STA INBUFF
B2DD A900 ; LDA #$0
B2DF 85B6 ; STA DATAD
B2E1 65F4 ; ADC INBUFF+1
B2E3 85F4 ; STA INBUFF+1
B2E5 90B8 B2A2 ; BCC :XRDI ; GO SCAN THIS NEXT LINE

B2E7 ; :XRD4
B2E7 A900 ; LDA #$0 ; CLEAR ELEMENT COUNT
B2E9 85F5 ; STA ZTEMP1

B2EB ; :XRD5
B2EB A5F5 ; LDA ZTEMP1
B2ED C5B6 ; CMP DATAD
B2EF B000 B2FC ; BCS :XRDI7 ; GET ELEMENT COUNT

B2F1 2005B3 ; :XRD6 JSR :XRNT ; AT PROPER ELEMENT
B2F4 D0F6 B2F1 ; BNE :XRDI6 ; BR IF AT
B2F6 B4B6 B2D0 ; BCS :XRDI3 ; ELSE SCAN FOR NEXT
B2FA D0EF B2EB ; BNE :XRDI5 ; GET CHAR

B2FC A940 ; :XRD7 LDA #$40 ; SET READ BIT
B2FE 85A6 ; STA DIRFLG
B300 66F2 ; INC CIX ; INC OVER DATA TOKEN

212
Source Code

B302 4C35B3 JMP :XINA ; GO DO IT

B305 B6F2 INC CIX ; INC INDEX
B307 A4F2 :XRNT1 LDY CIX ; GET INDEX
B309 B1F3 LDA [INBUFF],Y ; GET CHAR COUNT
B30B C92C CMP #$2C ; IS IT A COMMA
B30D 18 CLC ; CARRY CLEAR FOR COMMA
B30F F082 'B312 BEQ :XRNT2 ; BR IF COMMA
B310 C99B CMP #CR ; IS IT CR
B312 60 :XRNT2 RTS

B313 2034B9 :XROOD JSR ERROOD

XINPUT — Execute INPUT

B316 XINPUT

B316 A93F LDA #$'?' ; SET PROMPT CHAR
B318 85C2 STA PROMPT ; GET FIRST TOKEN
B31A 203EAB JSR GETTOK ; GET INPUT LINE
B31C 204EB3 JSR :XITB ; TEST BREAK
B31F 84A6 LDY #$0 ; SET INPUT MODE
B321 20F2B6 JSR GIOPRM ; SET CIX=0
B323 2051DA JSR :XRNT ; GET DEVICE NUM
B325 B326 :XIN0 ; SET DEVICE NO.
B326 2051DA JSR INTLBF ; GET END TOKEN
B328 2089BA JSR :XITB ; BACK UP OVER IT
B32A 84A6 STY DIRFLG ; IF NOT OPERATOR
B32C 203EAB JSR :XRNT ; GO GET DEVICE NUM
B32E 2051DA JSR :XRNT ; SET DEVICE NO.

B338 85A6 STA STINDEX ; INC OVER TOKEN
B33A 2002BD JSR GIOPRM ; SET CIX=0
B33C B332 :XIN0

B336 2051DA JSR INTLBF ; GET INPUT LINE
B338 2089BA JSR :XITB ; TEST BREAK
B33A 84A6 LDY #$0 ; SET INPUT MODE
B33C 203EAB JSR GETTOK ; GET TOKEN
B33E 2002BD JSR GIOPRM ; GET DEVICE NUM
B340 B342 :XIN0

B344 2002BD JSR GIOPRM ; SET DEVICE NO.
B346 2051DA JSR :XRNT ; GET END TOKEN
B348 2016AC JSR RTNVAR ; ERROR IF NO CR OR COMMA
B34A 4CB9B3 JMP :XINX ; RETURN VAR
B34C 20049A LDA #$0 ; GO FIGURE OUT WHAT TO DO NEXT
B34E B342 :XIN0

B350 D001 'B354 JSR TSTBRK ; GO TEST BREAK
B352 0008 BA BNE XITBT ; BR IF BRK
B354 4C93B7 XITBT JMP XSTOP ; STOP
B356 A990 JSR :XITB ; DONE
B358 85B4 STA ENTDTD ; ENTER DVC
B35A 2030B9 JSR ERRINP ; GO ERROR

B35E XISTR

B35E 202EAB JSR EXPINT ; INIT EXECUTE EXPR
B360 200005 JSR ARGPUSH ; PUS THE STRING
B362 C6F2 STA CIX ; DEC CIX TO CHAR
B364 A5F2 STA CIX ; BEFORE SOS
B366 85F5 STA 2TEMP1 ; SAVE THAT CIX
B368 B2F2 STA TEMP2 ; SET CHAR COUNT = -1
B36A 2A4F STA #$FF ; INC CHAR COUNT
B36C 80 :XIS1 INX ; INC CHAR COUNT
B36E 2005B3 JSR :XRNT ; GET NEXT CHAR
B370 D0FA 'B36C BNE :XIS1 ; BR NOT CR OR COMMA
B372 B004 'B378 BCS :XIS2 ; BR IF CR
B374 2A46 BIT DIRFLG ; IS IT COMMA, IF NOT READ
B376 50F4 'B36C BVC :XIS1 ; THEN CONTINUE

213
Source Code

B378 A4F5 ;xis2 Ldy ztemp1 ; get saved index
B37A A5A8 Lda stindex ; save index
B37C 48 Pha
B37D 8A TXA ; acu = char count
B37E A2F3 Ldx #inbuff ; point to inbuff
B380 2864AB JSR RISC ; go make str var
B383 60 PLA
B384 85A8 STA stindex ; restore index
B386 28A6AE JSR RIASC ; then do sta assign
B389 24A6 ; xinx bit dirflag ; is this read
B38B 50B7 "b39C LDA bvc :xinx ; br if not
B38D 6866 INC datad ; inc data displ
B38F 281DB9 JSR TSEND ; test end read stmt
B392 B0DD"B3AA BCS :XIRTS ; br if read end
B394 2007B3 ; xir1 JSR :XRTN1 ; get end data char
B397 9018"B3B1 BCC :XINC ; br if comma
B399 4CDBD2 JMP :XRD3 ; go get next data line
B39C ; xinx
B39C 281DB9 JSR TSEND
B39F 9008"B3AC BCS :XINC ; br branch
B3A1 2051DA ; xirto JSR INTLBF ; restore lbuff
B3A4 A900 LDA #0 ; restore enter
B3A6 85B4 STA ENTDTD ; device to zero
B3A8 60 RTS ; done
B3A9 2807B3 ; xin1 JSR :XRTN1 ; if not end of data
B3AC 9008"B3B1 BCC :XINC ; then branch
B3AE 4C26B3 JMP :XIN0 ; and continue
B3B1 68F2 ; xinc inc cix ; inc index
B3B3 4C35B3 JMP :XINA ; and continue

XPRINT — Execute PRINT Statement

B3B6 XPRINT
B3B8 A5C9 LDA PTABW ; get tab value
B3B9 85AF STA SCAT ; scant
B3BA A900 LDA #0 ; set out index = 0
B3BB 8594 STA COX
B3C0 A4A8 ; xpr0 Ldy stindex ; get stmt displ
B3C0 B18A LDA [STMCUR],Y ; get token
B3CC C912 CMP #ccom
B3C4 F053 "B419 BEQ :XPTAB ; br if tab
B3C6 C916 CMP #CCR
B3C8 F07C "B446 BEQ :XPEOL ; br if eol
B3CA C914 CMP #CEOS
B3CC F078 "B446 BEQ :XPEOL ; br if eol
B3CE C915 CMP #CSC
B3D0 F06F "B441 BEQ :XPNUL ; br if null
B3D2 C91C CMP #CPND
B3D4 F061 "B437 BEQ :XPRIOD
B3D6 2800AA JSR EXEPR ; go evaluate expression
B3D9 20F2AB JSR ARGPOP ; pop final value
B3DC C6A8 DEC STINDEX ; dec stindex
B3DE 24D2 BIT VTYPE ; is this a string
B3E0 3016 "B3F8 BMI :XPSTR ; br if string
B3E2 20E6D0 JSR CVFASC ; convert to ascii
B3E5 A900 LDA #0
B3E7 85F2 STA CIX
B3E9 A4F2 ; xpr1 Ldy cix ; output ascii characters

214
Source Code

B3EB B1F3 LDA [INBUFF],Y ; FROM INBUFF
B3ED 48 PLA ; UNTIL THE CHAR
B3EE E6F2 INC CIX ; WITH THE MSB ON
B3F0 2B5DB4 JSR :XPRC ; IS FOUND
B3F3 68 PLA
B3F4 1BF3 "B3E9 BPL :XPR1
B3F6 3BC6 "B3BE BMI :XPR0 ; THEN GO FOR NEXT TOKEN
B3F8 2B9NAB JSR GSTRAD ; GO GET ABS STRING ARRAY
B3FB A000 LDA $0
B3FD 85F2 STA CIX
B3FF A5D6 :XPR2C LDA VTYPE+EVSLEN ; IF LEN LOW
B401 D004 "B407 BNE :XPR2B ; NOT ZERO BR
B403 2C6D7 DEC VTYPE+EVSLEN+1 ; DEC LEN HI
B405 3B7 "B3BE BMI :XPR0 ; BR IF DONE
B407 2C6D6 :XPR2B DEC VTYPE+EVSLEN ; DEC LEN LOW

B409 A4F2 :XPR2 LDY CIX ; OUTPUT STRING CHARS
B40B 1D4 LDA [VTYPE+EVSADR],Y ; FOR THE LENGTH
B40D E6F2 INC CIX ; OF THE STRING
B40F D002 "B413 BNE :XPR2A
B411 E6D5 INC VTYPE+EVSADR+1
B413 :XPR2A
B416 2B5FB4 JSR :XPRC1
B419 4CFFB3 JMP :XPR2C ;

B419 A494 :XPTAB LDY COX ; DO UNTIL COX+1 < SCANT
B41B CB INY
B41C C4AF CPY SCANT
B41E 9009 "B429 BCC :XPR4
B420 1B :XPCI3 CLC
B421 A5C9 LDA PTABW ; SCANT = SCANT+TAB
B423 65AF ADC SCANT
B425 85AF STA SCANT
B427 9B80 "B419 BCC :XPR3

B429 A494 :XPR4 LDY COX ; DO UNTIL COX = SCANT
B42B C4AF CPY SCANT
B42D B012 "B441 BCS :XPR4A
B42F A920 LDA #$20 ; PRINT BLANKS
B431 2B5DB4 JSR :XPRC
B434 4C29B4 JMP :XPR4

B437 2B02BD :XPRGDD JSR GIOFRM ; GET DEVICE NO.
B43A 85B5 STA LISTDTD ; SET AS LIT DEVICE
B43C 65AB DEC STINDEX ;DEC INDEX
B43E 4CBBE3 JMP :XPR0 ; GET NEXT TOKEN

B441 ;
B441 E6A8 :XPNULL INC STINDEX ; INC STINDEX
B443 4CBBE3 JMP :XPR0

B446 ;
B446 A4AB :XEPEOL LDY STINDEX ; AT END OF PRINT
B448 88 DEY
B449 B18A LDA [STMCUR],Y ; IF PREV CHAR WAS
B44B C915 CMP #$C ; SEMI COLON THEN DONE
B44D F009 "B458 BEQ :XPRTN ; ELSE PRINT A CR
B44F C912 CMP #$COM ; OR A COMMA
B451 F005 "B458 BEQ :XPRTN ; THEN DONE
B453 A99B LDA #$CR
B455 2B5FB4 JSR :XPRC1 ; THEN DONE
B456 :XPRTN
B458 A900 LDA #$0 ; SET PRIMARY
B45A 85B5 STA LISTDTD ; LIST DVC = $0
B45C 60 RTS ; AND RETURN

B45D 2B9F :XPRC AND #$7F ; MSB OFF
B45F E694 :XPRC1 INC COX ; INC OUT INDEX
Source Code

XLPRINT — Print to Printer

XLPRINT  JMP PRCHAR ; OUTPUT CHAR

XLPRINT  LDA #PSTR&255 ; POINT TO FILE SPEC
           STA INBUFF  ; X
           LDA #PSTR/256 ; X
           STA INBUFF+1 ; X

LDX #7 ; GET DEVICE
STX LISTUD ; SET LIST DEVICE
LDA #0 ; GET AUX 2
LDY #8 ; GET OPEN TYPE

JSR SOPEN ; DO OPEN
JSR IOTEST ; TEST FOR ERROR
JSR XPRINT ; DO THE PRINT

JSR CLSYS1 ; CLOSE DEVICE

XLIST — Execute LIST Command

XLIST  LDA #0 ;SET TABLE SEARCH LINE NO
           STY TSLNUM ;TO ZERO
           STY TSLNUM+1

DEY
STY LELNUM ; SET LIST END LINE NO
LDA #$7F ;TO $7FF
STA LELNUM+1

STA $2FE ; SET NON-DISPLAY MODE
LDA #CR ; POINT CR

JSR FCHAR

JSR XGS ; SAVE CURLINE VIA GOSUB

LDY STINDEX ;GET STMT INDEX

INY ;INC TO NEXT CHAR

CPY NXTSTD ;RT NEXT STMT

BCS iLSTART ; BR IF AT, NO PARMS

LDY STINDEX ; SAVE STINDEX

PHA ; ON STACK

JSR POP1 ; POP FIRST ARGUMENT

PLA ; RESTORE STINDEX TO

STA STINDEX ; RE-DO FIRST ARG

LDA VTYPE ; GET VAR TYPE

BPL :XL1 ; BR IF NOT FILE SPEC STRING

JSR FLIST ; GO OPEN FILE

JSR iXL0 ; GO BACK TO AS IF FIRST PARM

BPL iXL1 ; BR IF AT, NO PARMS

JSR GETPINT ; GO GET START LNO

LDA AND ; POP FIRST ARGUMENT

LDA TSLNUM+1 ; MOVE START LNO

STA TSLNUM ;TO TSLNUM

LDX TSLNUM ; GET STMT INDEX

CPY NXTSTD ; AT NEXT STMT

BEQ iLSE ; BR IF AT, NO PARMS
B4C4 20D5AB JSR GETPINT ; GO GET LINE NO
B4C7 A5D4 :LSE LDA PR0 ; MOVE END LINE NO
B4C9 83AD STA LELENUM ; TO LIST END LINE NO
B4CB A5D5 LDA PR0+1 ;
B4CD 83AE STA LELENUM+1 ;
B4CF :LSTART
B4CF 20A2A9 JSR GETSTMT ; GO FIND FIRST LINE
B4D2 20E2A9 :LNXT JSR TENDST ; AT END OF STMTS
B4D5 3024 AB4FB STA LELNUM ; TO LIST
B4D7 A001 :L4RTN
B4D9 B18A LDA [STMCUR],Y ; LINE NO WITH END
B4DB C5AB CMP LELENUM+1 ; LINE NO
B4DD 900B ^B4EA BCC :LGO
B4DF D01A ^B4FB BNE :LRNT ; BR AT END
B4E2 B18A LDA [STMCUR],Y ; COMPARE CURRENT STMT
B4E4 C5AD CMP LELENUM ; LGO
B4E6 9002 ^B4EA BCC :LGO
B4E8 D011 ^B4FB BNE :LRNT ;
B4EA 205CB5 :LGO JSR :LLINE ; GO LIST THE LINE
B4ED 20F4A9 JSR TSTBRK ; TEST FOR BREAK
B4F0 20DDA9 JSR GETLL ; BR IF BREAK
B4F2 20DDA9 JSR GNXTL ; GO INC TO NEXT LINE
B4F8 4CD2B4 JMP :LNXT ; GO DO THIS LINE
B4FB :LRNT
B4FB A5B5 LDA LISTDT ; IF LIST DEVICE
B4FD F007 A B 506 BEQ :LRTNl ; IS ZERO, BR
B4FF 20F1BC JSR CLSYSD ; ELSE CLOSE DEVICE
B502 A900 LDA #0 ; AND RESET
B504 85B5 STA LISTDT ; DEVICE TO ZERO
B506 :LRTN1
B506 8DFE02 STA $2FE ; SET DISPLAY MODE
B509 4C19B7 JMP XRTN ; THEN RESTORE LIST LINE
B50C 86AA STX SRCISK ; SAVE SKIP LENGTH
B50E 2030B5 JSR :LSST ; SAVE SRC ADR
B511 A4AA :LSCI LDY SRCISK ; GET SKIP FACTOR
B513 C6AF DEY SCANT ; DECREMENT SRC COUNT
B515 300E ^B525 BMI :L5INC ; BR IF DONE
B517 B195 :LSC1 LDA [SRCADR],Y ; GET CHARACTER
B519 3003 ^B51E BMI :L5C2 ; BR IF LAST CHARACTER
B51B C8 INY ; INC TO NEXT
B51C D0F9 ^B517 BNE :L5C1 ; BR ALWAYS
B51E C8 :L5C2 INY
B51F 2025B5 JSR :LSINC ; INC SRC ADR BY Y
B522 4C1IB5 JMP :LSIC ; GO TRY NEXT
B525 18 :L5INC CLC ; Y PLUS
B526 98 TYA
B527 6595 ADC SRCADR ; SRCADR
B529 8595 STA SRCADR ; IS

LSCAN — Scan a Table for LIST Token

; ENTRY PARMS
; X = SKIP LENGTH
; A,Y = TABLE ADR
; SCANT = TOKEN
; B50C :LSCAN
B50C 86AA STX SRCISK ; SAVE SKIP LENGTH
B50E 2030B5 JSR :LSST ; SAVE SRC ADR
B511 A4AA :LSCI LDY SRCISK ; GET SKIP FACTOR
B513 C6AF DEY SCANT ; DECREMENT SRC COUNT
B515 300E ^B525 BMI :L5INC ; BR IF DONE
B517 B195 :LSC1 LDA [SRCADR],Y ; GET CHARACTER
B519 3003 ^B51E BMI :L5C2 ; BR IF LAST CHARACTER
B51B C8 INY ; INC TO NEXT
B51C D0F9 ^B517 BNE :L5C1 ; BR ALWAYS
B51E C8 :L5C2 INY
B51F 2025B5 JSR :LSINC ; INC SRC ADR BY Y
B522 4C1IB5 JMP :LSIC ; GO TRY NEXT
B525 18 :L5INC CLC ; Y PLUS
B526 98 TYA
B527 6595 ADC SRCADR ; SRCADR
B529 8595 STA SRCADR ; IS
Source Code

B52B A8 TAY ; NEW
B52C A596 LDA SRCADR+1 ; SRCADR
B52E 6900 ADC $0
;
B530 8596 :LSST STA SRCADR+1 ; STORE NEW SRCADR
B532 8495 STY SRCADR ; AND
B534 60 RTS ; RETURN

LPRTOKEN — Print a Token
B535 LPRTOKEN
B535 :LPRTOKEN
B535 A0FF LDY #$FF ; INITIALIZE INDEX TO ZERO
B537 84AF STY SCANT ;
;
B539 E6AF :LP1 INC SCANT ; INC INDEX
B53B A4AF LDY SCANT ; GET INDEX
B53D B195 LDA [SRCADR],Y ; GET TOKEN CHAR
B53F 49 PHA ; SAVE CHAR
B540 C99B CMP $CR ; IF ATARI CR
B542 F004 *B548 BEQ :LP1A ; THEN DON'T AND
B544 297F AND #$7F ; TURN OFF MSB
B546 F003 'B54B EQ :LP2 BR IF NON-PRINTING
B548 :LP1A
B54A B09FBA JSR PRCHAR GO PRINT CHAR
B54B :LP2
B54B 68 PLA ; GET CHAR
B54C 109F 'B539 BPL :LP1 ; BR IF NOT END CHAR
B54E 60 RTS ; GO BACK TO MY BOSS

LPTWB — Print Token with Blank Before and After
B54F :LPTWB
B54F A920 LDA #$20 ; GET BLANK
B551 209FBA JSR PRCHAR GO PRINT
B554 2035B5 :LPBT JSR :LPRTOKEN ; GO PRINT TOKEN
B557 A920 :LPBL LDA #$20 ; GET BLANK
B559 4C9FBA JMP PRCHAR GO PRINT IT AND RETURN

LLINE — List a Line
B55C LLINE
B55C :LLINE
B55C A000 LDY $0
B55E B18A LDA [STMCUR],Y ; MOVE LINE NO
B560 85D4 STA FR0 ; TO FR0
B562 8C INY
B563 B18A LDA [STMCUR],Y
B565 85D5 STA FR0+1
B567 20AADD JSR CVIFP ; CONVERT TO FP
B56A 20E6D8 JSR CVFASC ; CONVERT TO ASCII
B56D A5F6 LDA INBUFF ; MOVE INBUFF ADDR
B56F 8595 STA SRCADR ; TO SRCADR
B571 A5F4 LDA INBUFF+1
B573 8596 STA SRCADR+1
B575 2054B5 JSR :LPPTB ; AND PRINT LINE NO
;
B578 LDDLINE
B578 A002 LDY $2
B57A B18A LDA [STMCUR],Y ; GET LINE LENGTH
B57C 859F STA LENGTH ; AND SAVE
B57E 8C INY
B57F B18A :LL1 LDA [STMCUR],Y ; GET STMT LENGTH
B581 85A7 STA NXTSTD ; AND SAVE AS NEXT ST DISPL
B583 8C INY ; INC TO STMT TYPE
B584 84AB STY STINDEX ; AND SAVE DISPL
B586 2890B5 JSR :LSTMT ; GO LIST STMT
Source Code

B589 A4A7  LDY  NXTSTD  ; DONE LINE
B58B C49F  CPY  LNGTH
B58D 90F0  'B57F  BCC  :LL1  ; BR IF NOT
B58F 60  RTS  ; ELSE RETURN

LSTMT — List a Statement

B590 2831B6  JSR  :LGCT  ; GET CURRENT TOKEN
B593 C936  CMP  #CILET  ; IF IMP LET
B595 F817  'B5AE  BEQ  :LADV  ; BR
B597 283DB6  JSR  LSTMC  ; GO LIST STMT CODE

B59A 2831B6  JSR  :LGCT  ; GET CURRENT TOKEN
B59D C937  CMP  #CERR  ; BR IF ERROR STMT
B59F F004  'B5A5  BEQ  :LDR  ; WAS IT DATA OR REM
B5A1 C902  CMP  #2  ; BR IF NOT
B5A3 B009  'B5AE  BCS  :LADV  ; OUTPUT DATA/REM

B5A5 282FB6  :LDR  JSR  :LGNT  ; THEN PRINT THE CR
B5A8 280FBA  JSR  PRCHAR
B5AB 4CAB5  JMP  :LDR

B5AE 282FB6  :LADV  JSR  :LGNT  ; GET NEXT TOKEN
B5B1 101A  'B5CD  BFL  :LNVAR  ; BR IF NOT VARIABLE

B5B3 297F  AND  #$7F  ; TURN OFF MSB
B5B5 85AF  STA  SCANT  ; AND SET AS SCAN COUNT
B5B7 A200  LDX  #0  ; SCAN VNT FOR
B5B9 A5B3  LDA  VNTP+1  ; VAR NAME
B5BB A4B2  LDY  VNTP
B5BD 200CB5  JSR  :LSCAN

B5C0 2835B5  :LS1  JSR  :LPTOKEN  ; PRINT VAR NAME
B5C3 C9A8  CMP  #$AB  ; NAME END IN LPAREN
B5C5 D0E7  'B5AE  BNE  :LADV  ; DON'T PRINT NEXT TOKEN
B5C7 282FB6  JSR  :LGNT  ; IF IT IS A PAREN
B5CA 4CAB5  JMP  :LADV

B5CD  :LNVAR
B5CD C90F  CMP  #$8F  ; TOKEN: $8F
B5CF F81B  'B5E9  BEQ  :LSTC  ; BR IF OP, STR CONST

B5D1 B036  'B609  BCS  :LOP  ; BR IF TOKEN >$8F

B5D3 2840AB  JSR  NCTOFRO  ; ELSE IT'S NUM CONST
B5D6 C6AB  DEC  STINDEX  ; GO MOVE FR0
B5DB E6D8  JSR  CVFASC  ; BACK INDEX TO LAST CHAR
B5D8 8595  STA  SRCADR  ; CONVERT FR0 TO ASCII
B5DF 85A4  LDA  INBUFF  ; POINT SCRADR
B5E1 8596  STA  SRCADR+1  ; TO INBUFF WHERE
B5E3 2835B5  :LSX  JSR  :LPTOKEN  ; CHAR IS
B5E6 4CAB5  JMP  :LADV  ; GO PRINT NUMBER

B5E9 282FB6  :LSTC  JSR  :LGNT  ; GET NEXT TOKEN
B5EC 85AF  STA  SCANT  ; WHICH IS STR LENGTH
B5EE A922  LDA  #$22  ; PRINT DOUBLE QUOTE CHAR
B5F0 289FBA  JSR  PRCHAR
B5F3 A5AF  LDA  SCANT
B5F5 F00A  'B601  BEQ  :LS3  ; UNTIL COUNT =0

B5F7 282FB6  :LS2  JSR  :LGNT  ; OUTPUT STR CONST
B5FA 289FBA  JSR  PRCHAR  ; CHAR BY CHAR
B5FD C6AF  DEC  SCANT
B5FF DB0F  'B5F7  BNE  :LS2  ; UNTIL COUNT =0

B601  :LS3
B601 A922  LDA  #$22  ; THEN OUTPUT CLOSING
B603 289FBA  JSR  PRCHAR  ; DOUBLE QUOTE
B606 4CAEB5  JMP  :LADV
B609 38 :LOP SEC
B60A E910 SBC #$10 ; SUBTRACT THE 10
B60C 85AF STA SCANT ; SET FOR SCAN COUNT
B60E A200 LDX $0
B610 A9A7 LDA #$OPNTAB/256
B612 A983 LDY #$OPNTAB&255
B614 200CB5 JSR :LSCAN ; SCAN OP NAME TABLE
B617 2031B6 JSR :LGCT ; GET GO CURRENT TOKEN
B61A C93D CMP #$CPFUN ; IS IT FUNCTION
B61C B0C5 _B5E3 BCS :LSX ; BR IF FUNCTION
B61E A000 LDY $0
B620 B195 LDA [SRCADR],Y ; GET FIRST CHAR
B622 297F AND #$7F ; TURN OFF MSB
B624 20F7A3 JSR :STALPH ; TEST FOR ALPHA
B627 B08A _B5E3 BCS :LSX ; BR NOT ALPHA
B629 204F5B JSR :LPTWB ; LIST ALPHA WITH
B62C 4CAEB5 JMP :LADV ; BLANKS FOR AND AFTER
;
B62F 0 :LGNT ; GET NEXT TOKEN
B62F 66A8 INC STINDEX ; INC TO NEXT
B631 A4A8 JSR :DISPL ; GET DISPL
B633 C4A7 CPY NXTSTD ; AT END OF STMT
B635 B003 _B63A BCS :LGNT ; BR IF AT END
B637 B18A LDA [STMCUR],Y ; GET TOKEN
B637 68 RTS ; AND RETURN
;
B63A 68 :LGNT PLA ; POP CALLERS ADR
B63B 68 PLA ; AND
B63C 68 RTS ; GO BACK TO LIST LINE
;
B63D LSTMC
B63D 85AF STA SCANT ; SET INSSCAN COUNT
B63F A202 LDX $2 ; AND
B641 A9A4 LDA #$SNTAB/256 ; STATEMENT NAME TABLE
B643 A0A8 LDY #$SNTAB&255 ; STATEMENT NAME TABLE
B644 200CB5 JSR :LSCAN ; GO LIST WITH FOLLOWING BLANK
B645 4C54B5 JMP :LPTTB ; GO LIST WITH FOLLOWING BLANK

XFOR — Execute FOR

B64B LOCAL
B64B EXFOR
B64B 208ABB JSR :SAVDEX ; SAVE STINDEX
B64E 20E0AA JSR EXEXPR ; DO ASSIGNMENT
B651 A5D3 LDA VNUM ; GET VARIABLE #
B653 09B0 ORA #$80 ; OR IN HIGH ORDER BIT
B655 4B PHA
B656 2025BB JSR PIEXRSTK ; FIX RUN STACK

BUILD STACK ELEMENT

B659 A90C LDA #$FBDHY ; GET # OF BYTES
B65B B07B8B JSR :REXPAN ; EXPAND RUN STACK
B65E B00FAC JSR POP1 ; EVAL EXP & GET INTO FR0

PUT LIMIT [INFR0] ON STACK

B661 A2D4 LDX #FR0 ;POINT TO FR0
B663 A000 LDY #FLIM ; GET DISPL
B665 200FBB JSR :MV6RS ; GO MOVE LIMIT

SET DEFAULT STEP

B668 2044DA JSR $FR0 ; CLEAR FR0 TO ZEROS
B66B 3801 LDA #1 ; GET DEFAULT STEP
B66D 85D5 STA FR0+1 ; SET DEFAULT STEP VALUE
B66F A940 LDA #$40 ; GET DEFAULT EXPONENT
B671 85D4 STA FR0 ; STORE
Source Code

; TEST FOR END OF STMT
B673 2010B9 JSR TSTEND ; TEST FOR END OF START
B676 B003 "B67B BCS :NSTEP ; IF YES, WE ARE AT END OF STMT
; ELSE GET STEP VALUE
B678 200FAC JSR POPl ; EVAL EXP & GET INTO FR0
B67B :NSTEP ;
; PUT STEP [IN FR0] ON STACK
B67B A2D4 LDX #$FR0 ; POINT TO FR0
B67D A006 LDY #$STEP ; GET DISPL
B67F :200FBB JSR :MV6BB ; GC MOVE STEP
; PLA ; GET VARIABLE #
; PSHRSTK - PUSH COMMON PORT OF FOR/GOSUB
; - ELEMENT ON RUN STACK
; ON ENTRY A - VARIABLE # OR Ø [FOR GOSUB]
; TSLNUM - LINE #
; STINDEX - DISPL TO STMT TOKEN +1
B683 PSHRSTK ; EXPAND RUN STACK
B683 48 PHA ; SAVE VAR # / TYPE
B684 A904 LDA #$GFHEAD ; GET # OF BYTES TO EXPAND
B686 207BB8 JSR :REXPAN ; EXPAND [OLD TOP RETURN IN
; ZTEMP1]
; PLA ; GET VARIABLE #/TYPE
B689 68 LDA #$GTYPE ; GET DISPL TO TYPE IN HEADER
B68B A006 STA [TEMPA],Y ; PUT VAR#/TYPE ON STACK
; B688 B18A LDA [STEMCUR],Y ; GET LINE # LOW
B690 C8 INY ; POINT TO NEXT HEADER BYTE
B691 91C4 STA [TEMPA],Y ; PUT LINE # LOW IN HEADER
B693 B18A LDA [STEMCUR],Y ; GET LINE # HIGH
B695 C8 INY ; PUT IN HEADER
B696 91C4 STA [TEMPA],Y ;
; B698 A6B3 LDX SAVDEX ; GET SAVED INDEX INTO LINE
; B69A CA DEX ; POINT TO TOKEN IN LINE
B69B 8A TXA ; PUT IN A
B69C C8 INY ; POINT TO DISPL IN HEADER
B69D 91C4 STA [TEMPA],Y ; PUT IN HEADER
B69F 60 RTS

XGOSUB — Execute GOSUB
B6A0 XGOSUB X6A0 20C7B6 JSR XGS ; GO TO XGS ROUTINE

XGOTO — Execute GOTO
B6A3 XGOTO B6A3 20D5AB JSR GETPINT ; GET POSTIVE INTEGER IN FR0
; GET LINE ADRS & POINTERS
; XG02
B6A6 A5D5 LDA FR0+1 ; X
B6A8 85A1 STA TSLNUM+1 ; X
B6AA A5D4 LDA FR0 ; PUT LINE # IN TSLNUM
B6AC 85A0 STA TSLNUM ; X
Source Code

B6AE
; XGO1
B6AE 20A2A9 JSR GETSTM ; LINE POINTERS AND STMT ADDRESS
B6B1 B005 ~B6B8 BCS :ERLN ; IF NOT FOUND ERROR
B6B3 68 PLA ; CLEAN UP STACK
B6B4 68 PLA
B6B5 4C5FA9 JMP EXECNL ; GO TO EXECUTE CONTROL
B6B8 ;ERLN
B6B9 JSR RESCUR ; RESTORE STMT CURRENT
; ;
B6BB 202B99 JSR ERNOLN ; LINE # NOT FOUND
B6BE RESCUR
B6BE A5BE LDA SAVCUR ; RESTORE STMCUR
B6C0 85BA STA STMCUR
; X
B6C2 A5BF LDA SAVCUR+1
; X
B6C4 85BB STA STMCUR+1
; X
B6C6 60 RTS

XGS — Perform GOSUB [GOSUB, LIST, READ]
B6C7 XGS
B6C7 208AB8 JSR :SAVDEX ; GET STMT INDEX
B6CA XGS1
B6CA A900 LDA #0 ; GET GOSUB TYPE
B6CC 4C83B6 JMP PSHRSTK ; PUT ELEMENT ON RUN STACK

XNEXT — Execute NEXT
B6CF XNEXT
; GET VARIABLE #
; ;
B6CF A4A8 LDY STINDEX ; GET STMT INDEX
B6D1 B10A LDA [STMCUR],Y ; GET VARIABLE #
B6D3 85C7 STA ZTEMP2+1 ; SAVE
; ;
B6D5 ;XN
B6D5 2041B8 JSR PPRSTK ; FULL ELEMENT FROM RUN STACK
;VAR#/TYPE RETURN IN A
;B6D8 B03C ~B716 BCS :EREFOR ; IF AT TOP OF STACK, ERROR
;B6DA F03A ~B716 BEQ :EREFOR ; IF TYPE = GOSUB, ERROR
;B6DC C5C7 CMP ZTEMP2+1 ; DOES STKVAR# = OUR VAR #
;B6DE D0F5 ~B6D5 BNE :XN ; GET STEP VALUES IN FRI
; ;
;B6E0 A066 LDY $FSTEP ; GET DISPL INTO ELEMENT
B6E2 209EB8 JSR :PL66S ; GET STEP INTO FRI
; ;
;B6E5 A5E0 LDA FRI ; GET EXP FRI [CONTAINS SIGN]
B6E7 48 PHA ; PUSH ON CPU STACK
; ; GET VARIABLE VALUE
; ;
B6E8 A5C7 LDA ZTEMP2+1 ; GET VAR #
B6EA 2099AB JSR GETVAR ; GET VARIABLE VALUE
; ;
;B6ED 203BAD JSR FRADD ; ADD STEP TO VALUE
B6F0 2016AC JSR RTNVAR ; PUT IN VARIABLE TABLE
; ;
;GET LIMIT IN FRI
Source Code

B6F3 A000 LDY #FLIM ; GET DISPL TO LIMIT IN ELEMENT
B6F5 209EB8 JSR :PL6RS ; GET LIMIT INTO P1
B6F8 6B PLA ; GET SIGN OF STEP
B6F9 1006 "B701 BPL :STPPL ; BR IF STEP +

; COMPARE FOR NEGATIVE STEP

B6FB 2035AD JSR FRCMP ; COMPARE VALUE TO LIMIT
B6FE 1E09 "B709 BPL :NEXT ; IF VALUE > LIMIT, CONTINUE
B700 60 RTS ; ELSE DONE

; COMPARE FOR POSTIVE STEP

B701 :STPPL
B704 F003 "B709 BEQ :NEXT ; IF = CONTINUE
B706 3001 "B709 BMI :NEXT ; IF < CONTINUE
B708 60 RTS ; ELSE RETURN

; NEXT
B709 A910 LDA #GFHEAD+FBODY ; GET # BYTES IN FOR ELEMENT
B70B 2037B7 JSR :REXPAND ; GO PUT IT BACK ON STACK
B711 C908 CMP #CFOR IS TOKEN? ; IS TOKEN = FOR?
B713 D032 "B747 BNE :ERGFD IF NOT IT'S AN ERROR
B715 60 RTS

B716 :ERNOFOR
B716 2026B9 JSR ERNOFOR

XRTN — Execute RETURN

B719 XRTN
B719 2041B8 JSR POPRSTK ; GET ELEMENT FROM RUN STACK
B71C B016 "B734 BCS :ERRTN ; IF AT TOP OF STACK, ERROR
B71E D0F9 "B719 BNE XRTN ; IF TYPE NOT GOSUB, REPEAT

B720 2037B7 JSR :GETTOK ; GET TOKEN FROM LINE [IN A]
B723 C90C CMP #CGOSUB ; IS IT GOSUB?
B725 F00C "B733 BEQ :XRTS ; BR IF GOSUB
B727 C91E CMP #CON
B729 F008 "B733 BEQ :XRTS ; BR IF ON
B72B C904 CMP #CLIST
B72D F004 "B733 BEQ :XRTS ; BR IF LIST
B72F C922 CMP #CREAD
B731 D014 "B747 BNE :ERGFD ; IF NOT, ERROR
B733 :XRTS
B733 60 RTS

; ERRTN
B734 :ERRTN
B734 2020B9 JSR ERBRTN ; BAD RETURN ERROR

*:GETTOK — GET TOKEN POINTED TO BY RUN STACK ELEMENT

ON EXIT A — CONTAINS TOKEN

B735 :GETTOK
B737 2018B8 JSR SETLINE ; SET UP TO PROCESS LINE
B73A B008 "B747 BCS :ERGFD ; IF LINE # NOT FOUND, ERROR

; SDISP ; GET DISPL TO TOKEN
B73C A4B2 LDY SVDISP
B73E 88 DEY
B740 8B78 STA [STMCUR],Y ; GET NEXT STMT DISPL
B741 85A7 STA NXTSTD ; SAVE

; INY ; GET DISPL TO TOKEN AGAIN
B743 C8 INY
B744 B18A LDA [STMCUR],Y ; GET TOKEN
B746 60 RTS

B747 :ERGFD

223
Source Code

XRUN — Execute RUN

B74D 2010B9 JSR TSTEND ; CHECK FOR END OF STMT
B750 B003 \*B755 BCS :NOFILE ; IF END OF STMT, BR
B752 20F7BA JSR FRUN ; ELSE HAVE FILE NAME
B755 :NOFILE

GET 1ST LINE # OF PROGRAM

B757 A900 LDA #0 ; GET SMALLEST POSSIBLE LINE NUM
B759 85A0 STA TSLNUM ; X
B75B 85A1 STA TSLNUM+1 ; X
B75E 2010B8 JSR SETLINE ; SET UP LINE POINTERS
B761 3012 \*B775 BMI :RUNEND ; IF AT END, BR
B763 20F9B8 JSR RUNINIT ; CLEAR SOME STORAGE

FALL THRU TO CLR

XCLR — Execute CLR

B766 20C0B8 JSR ZVAR ; GO ZERO VARS
B769 20F2B8 JSR RSTPTR ; GO RESET STACK PTRS
B76C A900 LDA #0 ; CLEAR DATA VALUES
B76E 85B7 STA DATALN
B770 85B8 STA DATALN+1
B772 85B6 STA DATAD
B774 60 RTS

B775 :RUNEND
B775 4C50A0 JMP SNX1 ;NO PROGRAM TO RUN

XIF — Execute IF

B778 B00FAC JSR POP1 ; EVAL EXP AND GET VALUE INTO FR0
B77A 4A5D LDA FR0M ; GET 1ST MANTISSA BYTE
B77D F009 \*B780 BEQ :FALSE ; IF = 0, # = 0 AND IS FALSE

EXPRESSION TRUE

B77F 2010B9 JSR TSTEND ; TEST FOR END OF STMT
B780 B003 \*B787 BCS :TEOS ; IF AT EOS, BRANCH

TRUE AND NOT EOS

B784 4CA3B6 JMP XGOTO ; JOIN GOTO

TRUE AND EOS

B787 :TEOS
B787 60 RTS

EXPRESSION FALSE

B788 :FALSE
B788 A59F LDA LLNGTH ; GET DISPL TO END OF LINE
B78A 85A7 STA NXTSTD ; SAVE AS DISPL TO NEXT SYMT
B78C 60 RTS

224
Source Code

XEND — Execute END

```
B78D  XEND
B78D  20A7B7  JSR  STOP
B790  4C50A0  JMP  SNX1
```

XSTOP — Execute STOP

```
B793  XSTOP
B793  20A7B7  JSR  STOP  ; GO SET UP STOP LINE #
B793  PRINT MESSAGE  ;
B796  206EBD  JSR  PRCR  ; PRINT CR
B799  A9B6  LDA  #:MSTOP&255  ; SET POINTER FOR MESSAGE
B79B  8595  STA  SRCADR  ; X
B79D  A9B7  LDA  #:MSTOP/256  ; X
B79F  8596  STA  SRCADR+1  ; X
B7A1  2035B5  JSR  STOP  ;
B7A4  4C74B9  JMP  :ERRM2  ; PRINT REST OF MESSAGE
B7A7  STOP
B7A7  20E2A9  JSR  TENDST  ; GET CURRENT LINE # HIGH
B7AA  30E7 "B7B3  BMI  :STOPEND  ; IF -, THIS IS DIRECT STMT
B7AC  85BB  STA  STOPLN+1  ; SAVE LINE # HIGH FOR CON
B7AE  88  DEY  ; DEC INDEX
B7AF  B18A  LDA  [STMCUR],Y  ; GET LINE # LOW
B7B1  B58A  STA  STOPLN  ; SAVE FOR CON
B7B3  :STOPEND
B7B3  4C72BD  JMP  SETDZ  ; SET L/D DEVICE =0
B7B6  53544F5050  :MSTOP DC 'STOPPED' 4544A0
```

XCONT — Execute Continue

```
B7BE  XCONT
B7BE  20E2A9  JSR  TENDST  ; IS IT INDIRECT STMT?
B7C1  10F0 "B7B3  BPL  :STOPEND  ; IF YES, BR
B7C3  A5BA  LDA  STOPLN  ; SET STOP LINE # AS LINE #
B7C5  85A0  STA  TSLNUM  ; FOR GET
B7C7  A5BB  LDA  TSLNUM+1  ; X
B7C9  85A1  STA  TSLNUM+1  ; X
B7CB  20A2A9  JSR  GETSTMT  ; GET ADR OF STMT WE
B7CE  20E2A9  JSR  TENDST  ; STOPPED AT
B7D1  30A2 "B775  BMI  :RUNEND  ; AT END OF STMT TAB ?
B7D3  20DDA9  JSR  GETLL  ; GET NEXT LINE ADDR IN CURSTM
B7D6  20DDA9  JSR  GNXTL  ; X
B7D9  20E2A9  JSR  TENDST  ; SEE IF WE ARE AT END OF
B7DC  3097 "B775  BMI  :RUNEND  ; STMT TABLE
B7DE  4C1BB8  JMP  SETLNI  ; BR IF MINUS
B7E1  20E2AB  JSR  GETINT  ; SET UP LINE POINTERS
```

XTRAP — Execute TRAP

```
B7E1  XTRAP
B7E1  20E2AB  JSR  GETINT  ; CONVERT LINE # TO POSITIVE INT
B7E4  A5D4  LDA  FR0  ; SAVE LINE # LOW AS TRAP LINE
B7E6  85BC  STA  TRAPLN  ; IN CASE OF LATER ERROR
B7EB  A5D5  LDA  FR0+1  ; X
B7EA  85BD  STA  TRAPLN+1  ; X
B7EC  60  RTS
```
Source Code

XON — Execute ON

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>B7ED</td>
<td>JSR :SAUDEX</td>
<td>Save index into line</td>
</tr>
<tr>
<td>B7ED</td>
<td>LDA FR0</td>
<td>Get value</td>
</tr>
<tr>
<td>B7F3</td>
<td>BEQ :ERV</td>
<td>If zero, fall through to next stmt.</td>
</tr>
<tr>
<td>B7F7</td>
<td>LDY STINDEX</td>
<td>Get stmt index</td>
</tr>
<tr>
<td>B7F9</td>
<td>DEY</td>
<td>Back up to GSUB/GOTO</td>
</tr>
<tr>
<td>B7FA</td>
<td>LDA [STMCUR],Y</td>
<td>Get code</td>
</tr>
<tr>
<td>B7FC</td>
<td>CMP #CGTO</td>
<td>Is it GOTO?</td>
</tr>
<tr>
<td>B7FE</td>
<td>BEQ :GO</td>
<td>If yes, don't push on run stack</td>
</tr>
</tbody>
</table>

; This is on - GSUB: Put element on run stack

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>B800</td>
<td>JSR XGS1</td>
<td>Put element on run stack</td>
</tr>
<tr>
<td>B803</td>
<td>JSR XGSl</td>
<td>For return</td>
</tr>
<tr>
<td>B803</td>
<td>LDA FR0</td>
<td>Get index into expressions</td>
</tr>
<tr>
<td>B805</td>
<td>STA ONLOOP</td>
<td>Save for loop control</td>
</tr>
<tr>
<td>B807</td>
<td>JSR GETPINT</td>
<td>Get + integer</td>
</tr>
<tr>
<td>B80A</td>
<td>DEC ONLOOP</td>
<td>Is this the line # we want?</td>
</tr>
<tr>
<td>B80C</td>
<td>BEQ :ON2</td>
<td>If yes, go do it</td>
</tr>
<tr>
<td>B80E</td>
<td>JSR TSTEND</td>
<td>Are there more expressions?</td>
</tr>
<tr>
<td>B811</td>
<td>BCC :ON1</td>
<td>If yes, then eval next one</td>
</tr>
<tr>
<td>B813</td>
<td>RTS</td>
<td>Else fall through to next stmt.</td>
</tr>
<tr>
<td>B814</td>
<td>JMP XG02</td>
<td>Join GOTO</td>
</tr>
<tr>
<td>B817</td>
<td>RTS</td>
<td>Fall through to next stmt.</td>
</tr>
</tbody>
</table>

Execution Control Statement Subroutines

SETLINE — Set Up Line Pointers

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>B818</td>
<td>JSR GETSTM</td>
<td>Get STMCUR</td>
</tr>
<tr>
<td>B81B</td>
<td>SETLNM1</td>
<td></td>
</tr>
<tr>
<td>B81B</td>
<td>LDY #2</td>
<td>Get disp in line to length</td>
</tr>
<tr>
<td>B81D</td>
<td>LDA [STMCUR],Y</td>
<td>Get line length</td>
</tr>
<tr>
<td>B81F</td>
<td>STA LNGTH</td>
<td>Set line length</td>
</tr>
<tr>
<td>B821</td>
<td>INY</td>
<td>Point to next stmt displ</td>
</tr>
<tr>
<td>B822</td>
<td>STY NXTSTD</td>
<td>Set nxt stmt displ</td>
</tr>
<tr>
<td>B824</td>
<td>RTS</td>
<td></td>
</tr>
</tbody>
</table>

FIXRSTK — Fix Run Stack — Remove Old FORs

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>On entry A - variable # in current FOR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>On exit runstk clear of all for's</td>
</tr>
</tbody>
</table>
Source Code

B825 85C7  FIXRSTK STA ZTEMP2+1 ; SAVE VAR # OF THIS FOR
; ;
B827 2081B8  JSR :SAVRTOP ; SAVE TOP OF RUN STACK IN
; ;
B82A 041B8  JSR POPRSTK ; POP AN ELEMENT FROM RUNSTK
B82D 8008 'B837  BCS :TOP ; IF AT TOP - WE ARE DONE
B82F F006 'B837  BEQ :TOP ; IF CC = 0B ELEMENT WAS GOSUB
B831 C5C7  CMP ZTEMP2+1 ; IS STK VAR # = OUR VAR #?
B833 F00B 'B840  BEQ :FNVAR ; IF YES, WE ARE DONE
B835 D0F3 'B82A  BNE :FIXR ; ELSE LOOK AT NEXT ELEMENT
; ;
; ;
FOR VAR # NOT ON STACK ABOVE TOP OR GOSUB
[RESTORE TOP OF STACK]
; ;
B837 40C5  LDA TEMPA ; RESTORE TOPRSTK
B839 8590  STA TOPRSTK ; X
B83B A5C5  LDA TEMPA+1 ; X
B83D 8591  STA TOPRSTK+1 ; X
B83F 60  RTS
; ;
FOR VAR # FOUND ON STACK
; ;
B840 041B8  JSR :FNVAR
B840 60  RTS

POPRSTK — Pop Element from Run Stack

* ON EXIT A - TYPE OF ELEMENT OR VAR #
* X - DISPL INTO LINE OF FOR/GOSUB TOKEN
* CUSET - CARRY SET STACK WAS EMPTY
* CARRY CLEAR - ENTRY POPED
* EQ SET - ELEMENT IS GOSUB
* TSLNUM - LINE #

B841 041B8  XPOP POPRSTK
; ;
TEST FOR STACK EMPTY
; ;
B841 A5F8  LDA RUNSTK+1 ; GET START OF RUN STACK HIGH
B843 C591  CMP TOPRSTK+1 ; IS IT < TOP OF STACK HIGH
B845 9008 'B84F  BCC :NTOP ; IF YES, WE ARE NOT AT TOP
B847 A5E8  LDA RUNSTK ; GET START OF RUN STACK LOW
B849 C590  CMP TOPRSTK ; IS IT < TOP OF STACK LOW
B84B 9002 'B84F  BCC :NTOP ; IF YES, WE ARE NOT AT TOP
; ;
B84D 60  RTS ; ELSE AT TOP: SET CARRY
B84E 60  RTS ; RETURN
; ;
GET 4 BYTE HEADER
[COMMON TO GOSUB AND FOR]
; ;
B84F 041B8  JSR :RCONT ; TAKE IT OFF STACK
B851 2072B8  JSR :#GFHEAD ; GET LENGTH OF HEADER
B854 A083  LDA #GFDISP ; GET INDEX TO SAVED LINE DISPL
B856 B190  LDA [#TOPRSTK],Y ; GET SAVED LINE DISPL
B858 85B2  STA SVDISP ; SAVE
B85A 88  DEY ; POINT TO LINE # IN HEADER
B85B B190  LDA [#TOPRSTK],Y ; GET LINE # HIGH
B85D 85A1  STA TSLNUM+1 ; SAVE LINE # HIGH
B85F 88  DEY ; GET DISPL TO LINE # LOW

227
B860  B190  LDA [TOPRSTK],Y ; GET LINE # LOW
B862  B5A0  STA TSLNUM ; SAVE LINE # LOW

B864  80  DEY ; POINT TO TYPE
B865  B190  LDA [TOPRSTK],Y ; GET TYPE
B867  F007  B870  BEQ :FND ; IF TYPE = GOSUB, SET ELEMENT

GET 12 BYTE FOR BODY

B869  48  PHA ; SAVE VAR #
B86A  A90C  LDA #BODY ; GET # BYTES TO POP
B86C  2872B8  JSR :REXPAN ; POP FROM RUN STACK
B86E  68  PLA ; GET VAR #

B870  :FND  CLC CLEAR [ENTRY POPPED]
B871  68  RTS ; CLEAR CARRY [ENTRY POPPED]

:REXPAN — Expand Run Stack

* ON ENTRY A - # OF BYTES TO ADD
  ** ON EXIT ZTEMP1 - OLD TOPRSTK

B872  :REXPAN
B872  A8  TAY ; Y=LENGTH
B873  A290  LDY #TOPRSTK ; X = PTR TO RUN STACK
B875  4CFA8  JMP CONTLOW

:SAVRTOP — Save Top of Run Stack in ZTEMP1

B881  :SAVRTOP
B881  A690  LDY TOPRSTK ; SAVE TOPRSTK
B883  86C4  STX TEMPA ; X
B885  A691  LDX TOPRSTK+1 ; X
B887  86C5  STX TEMPA+1
B889  68  RTS

:SAVDEX — Save Line Displacement

B88A  :SAVDEX
B88A  A4A8  LDY STINDEX ; GET STMT INDEX
B88C  B4B3  STY SAVDEX ; SAVE IT
B88E  60  RTS

:MV6RS — Move 6-Byte Value to Run Stack

* ON ENTRY X - LOCATION TO MOVE FROM
  ** Y- DISPL FROM ZTEMP1 TO MOVE TO
  *** ZTEMP1 - LOCATION OF RUN STK ELEMENT

B88F  :MV6RS
B88F  A906  LDA #6 ; GET # OF BYTES TO MOVE
B891  85C6  STA ZTEMP2 ; SAVE AS COUNTER
B893  :MV
B893  B500  LDA 0,X ; GET A BYTE
B895  91C4  STA [TEMPS],Y ; PUT ON STACK
B897  E0  INX ; POINT TO NEXT BYTE
B898  C8  INY ; POINT TO NEXT LOCATION
B899  C6C6  DEC ZTEMP2 ; DEC COUNTER
B89B  B8F6  ~B893  BNE :MV ; IF NOT = 0 DO AGAIN
B89D  60  RTS

228
PL6RS — Pull 6 Bytes from Run Stack to FR1

* ON ENTRY  Y = DISPL FROM TOPRSTK TO MOVE FROM  
  TOPRSTK — START OF ELEMENT  
  
B90E  :PL6RS  
B90E A006  LDA #6  ; GET # OF BYTES TO MOVE  
B8A0  85C6  STA ZTEMP2  ; SAVE AS COUNTER  
B8A2  A2E0  LDX #FR1  
B8A4  :PL  
B8A4 B190  LDA [TOPRSTK],Y  ; GET A BYTE  
B8A6  9500  STA 0,X  ; SAVE IN Z PAGE  
B8A8  E8  INX  ; INC TO NEXT LOCATION  
B8A9  C8  INY  ; INC TO NEXT BYTE  
B8AA  C6C6  DEC ZTEMP2  ; DEC COUNTER  
B8AC  D8F6 *B8A4  BNE :PL  ; IF NOT =0, DO AGAIN  
B8AE  60  RTS  

RSTPTR — Reset StackPointers [STARP and RUNSTK]

*  
B8AF  RSTPTR  
B8AF A58C  LDA STARP  ; GET BASE OF STR/ARRAY  
  SPACE LOW  
B8A1  858E  STA RUNSTK  ; RESET  
B8A3  8580  STA MEMTOP  
B8A5  858E  STA APHM  ; SET APPLICATION HIMEM  
B8A7  A58D  LDA STARP+1  ; GET BASE STR/ARRAY SPACE  
  HIGH  
B8A9  858F  STA RUNSTK+1  ; RESET  
B8BB  8591  STA MEMTOP+1  ; X  
B8BD  858F  STA APHM+1  ; SET APPLICATION HIMEM  
B8BF  60  RTS  

ZVAR — Zero Variable

B8C0  ZVAR  
B8C0 A686  LDX VVTP  ; MOVE VARIABLE TABLE POINTER  
B8C2  86F5  STX ZTEMP1  ; X  
B8C4  A487  LDY VVTP+1  ; X  
B8C6  84F6  STY ZTEMP1+1  ; X  

; ARE WE AT END OF TABLE ?  
B8C8  :ZVAR1  
B8C8 A6F6  LDX ZTEMP1+1  ; GET NEXT VARIABLE ADDR HIGH  
B8CA  E489  CPX ENDVVT+1  ; IS IT < END VALUE HIGH  
B8CC  9007 *BBD5  BCC :ZVAR2  ; IF YES, MORE TO DO  
B8CE  A6F5  LDX ZTEMP1  ; GET NEXT VARIABLE ADDR LOW  
B8D0  E488  CPX ENDVVT  ; IS IT < END VALUE LOW  
B8D2  9001 *BBD5  BCC :ZVAR2  ; IF YES, MORE TO DO  
B8D4  60  RTS  ; ELSE, DONE  

; ZERO A VARIABLE  
B8D5  :ZVAR2  
B8D5 A000  LDY #0  ; TURN OFF  
B8D7  B1F5  LDA [ZTEMP1],Y  ; DIM FLAG  
B8D9  29F5  AND #SFE  
B8DB  91F5  STA [ZTEMP1],Y  
B8DD  A002  LDY #2  ; INDEX PAST VARIABLE HEADER  
B8DF  A206  LDX #6  ; GET # OF BYTES TO ZERO  
B8E1  A0B0  LDA #0  ; CLEAR A  
B8E3  :ZVAR3  
B8E3 91F5  STA [ZTEMP1],Y  ; ZERO BYTE  
B8E5  C8  INY  ; POINT TO NEXT BYTE  
B8E6  A6  DEX  ; DEC POINTER  
B8E7  D0FA *B8E3  BNE :ZVAR3  ; IF NOT = 0, ZERO NEXT BYTE  

ZVAR — Zero Variable  

B8C0  ZVAR  
B8C0 A686  LDX VVTP  ; MOVE VARIABLE TABLE POINTER  
B8C2  86F5  STX ZTEMP1  ; X  
B8C4  A487  LDY VVTP+1  ; X  
B8C6  84F6  STY ZTEMP1+1  ; X  

; ARE WE AT END OF TABLE ?  
B8C8  :ZVAR1  
B8C8 A6F6  LDX ZTEMP1+1  ; GET NEXT VARIABLE ADDR HIGH  
B8CA  E489  CPX ENDVVT+1  ; IS IT < END VALUE HIGH  
B8CC  9007 *BBD5  BCC :ZVAR2  ; IF YES, MORE TO DO  
B8CE  A6F5  LDX ZTEMP1  ; GET NEXT VARIABLE ADDR LOW  
B8D0  E488  CPX ENDVVT  ; IS IT < END VALUE LOW  
B8D2  9001 *BBD5  BCC :ZVAR2  ; IF YES, MORE TO DO  
B8D4  60  RTS  ; ELSE, DONE  

; ZERO A VARIABLE  
B8D5  :ZVAR2  
B8D5 A000  LDY #0  ; TURN OFF  
B8D7  B1F5  LDA [ZTEMP1],Y  ; DIM FLAG  
B8D9  29F5  AND #SFE  
B8DB  91F5  STA [ZTEMP1],Y  
B8DD  A002  LDY #2  ; INDEX PAST VARIABLE HEADER  
B8DF  A206  LDX #6  ; GET # OF BYTES TO ZERO  
B8E1  A0B0  LDA #0  ; CLEAR A  
B8E3  :ZVAR3  
B8E3 91F5  STA [ZTEMP1],Y  ; ZERO BYTE  
B8E5  C8  INY  ; POINT TO NEXT BYTE  
B8E6  A6  DEX  ; DEC POINTER  
B8E7  D0FA *B8E3  BNE :ZVAR3  ; IF NOT = 0, ZERO NEXT BYTE  

RSTPTR — Reset StackPointers [STARP and RUNSTK]

*  
B8AF  RSTPTR  
B8AF A58C  LDA STARP  ; GET BASE OF STR/ARRAY  
  SPACE LOW  
B8B1  858E  STA RUNSTK  ; RESET  
B8B3  8580  STA MEMTOP  
B8B5  858E  STA APHM  ; SET APPLICATION HIMEM  
B8B7  A58D  LDA STARP+1  ; GET BASE STR/ARRAY SPACE  
  HIGH  
B8B9  858F  STA RUNSTK+1  ; RESET  
B8BB  8591  STA MEMTOP+1  ; X  
B8BD  858F  STA APHM+1  ; SET APPLICATION HIMEM  
B8BF  60  RTS  

ZVAR — Zero Variable

B8C0  ZVAR  
B8C0 A686  LDX VVTP  ; MOVE VARIABLE TABLE POINTER  
B8C2  86F5  STX ZTEMP1  ; X  
B8C4  A487  LDY VVTP+1  ; X  
B8C6  84F6  STY ZTEMP1+1  ; X  

; ARE WE AT END OF TABLE ?  
B8C8  :ZVAR1  
B8C8 A6F6  LDX ZTEMP1+1  ; GET NEXT VARIABLE ADDR HIGH  
B8CA  E489  CPX ENDVVT+1  ; IS IT < END VALUE HIGH  
B8CC  9007 *BBD5  BCC :ZVAR2  ; IF YES, MORE TO DO  
B8CE  A6F5  LDX ZTEMP1  ; GET NEXT VARIABLE ADDR LOW  
B8D0  E488  CPX ENDVVT  ; IS IT < END VALUE LOW  
B8D2  9001 *BBD5  BCC :ZVAR2  ; IF YES, MORE TO DO  
B8D4  60  RTS  ; ELSE, DONE  

; ZERO A VARIABLE  
B8D5  :ZVAR2  
B8D5 A000  LDY #0  ; TURN OFF  
B8D7  B1F5  LDA [ZTEMP1],Y  ; DIM FLAG  
B8D9  29F5  AND #SFE  
B8DB  91F5  STA [ZTEMP1],Y  
B8DD  A002  LDY #2  ; INDEX PAST VARIABLE HEADER  
B8DF  A206  LDX #6  ; GET # OF BYTES TO ZERO  
B8E1  A0B0  LDA #0  ; CLEAR A  
B8E3  :ZVAR3  
B8E3 91F5  STA [ZTEMP1],Y  ; ZERO BYTE  
B8E5  C8  INY  ; POINT TO NEXT BYTE  
B8E6  A6  DEX  ; DEC POINTER  
B8E7  D0FA *B8E3  BNE :ZVAR3  ; IF NOT = 0, ZERO NEXT BYTE
Source Code

RUNINIT — Initialize Storage Locations for RUN

RUNINIT
BBF8 A000 LDY #$0
BBFA 849A STY STOPLN
BBFB 84BB STY STOPLN+1
BBFE 8489 STY ERRNUM
B900 84FB STY RADFLG
B902 8486 STY DATAD
B904 8487 STY DATALN
B906 8488 STY DATALN+1
B908 DEY
B909 84BD STY TRAPLN+1
B90B 8411 STY BRKBYTE
B90D 4C41BD JMP CLSALL

TSTEND — Test for End of Statement

ON EXIT CC SET
*CARRY CLEAR - END OF STMT
*CARRY SET - END OF STMT

B910 A6A8 LDX STINDEX
B912 E8 INX
B913 E4A7 CPX NXTSTD
B915 60 RTS

Error Message Routine

Error Messages

B916 E689 ERRNSF INC ERRNUM ; FILE NOT SAVE FILE
B918 E689 ERDNO INC ERRNUM ; #DN0 > 7
B91A E689 ERRTPL INC ERRNUM ; LOAD PGM TOO BIG
B91C E689 ERSVAL INC ERRNUM ; STRING NOT VALID
B91E E689 XOR INC ERRNUM ; EXECUTION OF GARBAGE
B920 E689 ERBRTN INC ERRNUM ; BAD RETURNS
B922 E689 ERGFDE INC ERRNUM ; GOSUB/FOR LINE DELETED
B924 E689 ERLTL INC ERRNUM ; LINE TO LONG
B926 E689 ERRFOR INC ERRNUM ; NO MATCHING FOR
B928 E689 ERNOLN INC ERRNUM ; LINE NOT FOUND [GOSUB/GOTO]
B92A E689 EROVFL INC ERRNUM ; FLOATING POINT OVERFLOW
B92C E689 ERRAGS INC ERRNUM ; ARG STACK OVERFLOW
B92E E689 ERRDIM INC ERRNUM ; ARRAY/STRING DIM ERROR
B930 E689 ERRINP INC ERRNUM ; INPUT STMT ERROR
B932 E689 ERBLN INC ERRNUM ; VALUE NOT <$32768
B934 E689 ERBRT INC ERRNUM ; READ OUT OF DATA
B936 E689 ERRSSL INC ERRNUM ; STRING LENGTH ERROR
B938 E689 ERRVSF INC ERRNUM ; VARIABLE TABLE FULL
B93A E689 ERVTR INC ERRNUM ; VALUE ERROR
B93C E689 MEMFULL INC ERRNUM ; MEMORY FULL
B93E E689 ERON INC ERRNUM ; NO LINE # FOR EXP IN ON
Source Code

Error Routine

1. **ERROR**
   - **B940**: LDA #0
   - **B940**: STA DSPFLG ; FLAG
   - **B942**: JSR STOP ; SET LINE # STopped AT
   - **B948**: LDA TRAPLN+1 ; GET TRAP LINE # HIGH
   - **B94A**: J SR :ERRM1 ; IF NO LINE # PRINT MESSAGE

   * * *
   - **B94C**: STA TSLNUM+1 ; SET TRAP LINE # HIGH FOR
   - **B94E**: LDA TSLNUM ; GET FOR GET STMT
   - **B952**: LDA #$80 ; TURN OFF TRAP
   - **B954**: STA TRAPLN+1
   - **B956**: LDA ERRNUM ; GET ERROR #
   - **B958**: STA ERRSAV ; SAVE IT
   - **B95A**: STA #0 ; CLEAR
   - **B95C**: STA ERRNUM ; ERROR#
   - **B95E**: LDA #0 ; CLEAR

   * * *
   - **B961**: J SR :ERRM1 ; TRAP SET - GO TO SPECIFIED LINE #

   * * *
   - **B961**: LDA TRAPLN+1 ; GET TRAP LINE # LOW
   - **B963**: STA TSLNUM ; SET FOR GET STMT
   - **B965**: LDA ERRNUM ; GET ERROR #
   - **B967**: STA ERRSAV ; SAVE IT
   - **B96F**: STA #0 ; CLEAR
   - **B971**: LDA TRAPLN ; GET TRAP
   - **B973**: STA TSLNUM ; SET FOR GET STMT
   - **B975**: LDA ERRNUM ; ERROR#

Print Error Message Part 1 [**ERR]**

1. **B961**: JSR PRCR ; PRINT CR
2. **B964**: JSR #CERR ; GET TOKEN FOR ERROR
3. **B966**: JSR LSTMC ; GO PRINT CODE

Print Error Number

1. **B969**: LDA ERRNUM ; GET ERROR #
2. **B96B**: STA FR0 ; SET ERROR # OF FR0 AS INTEGER
3. **B96D**: LDA #0 ; SET ERROR # HIGH
4. **B96F**: STA FR0+1 ; X

   * *
   - **B971**: JSR :PRINUM ; GO PRINT ERROR #

   * *
   - **B974**: JSR TENDST ; TEST FOR DIRECT STMT
   - **B977**: JSR :ERRDONE ; IF DIRECT STMT, DONE

Print Message Part 2 [AT LINE]

1. **B979**: LDA #:ERRMS&255 ; SET POINTER TO MSG FOR PRINT
2. **B97B**: LDA SRCADR ; X
3. **B97D**: LDA #:ERRMS/256 ; X

   * *
   - **B97F**: LDA SRCADR+1 ; X

   * *
   - **B981**: JSR LPRTOKEN ; PRINT LINE

Print Line Number

1. **B984**: LDA [STMCUR],Y ; GET LINE # HIGH
2. **B98B**: STA FR0+1 ; SET IN FR0 FOR CONVERT
3. **B989**: STA FR0 ; GET CURRENT LINE # LOW
4. **B98B**: LDA [STMCUR],Y ; GET UNUSED LINE # LOW
5. **B98D**: STA FR0 ; SET IN FR0 LOW FOR CONVERT

   * *
   - **B98F**: JSR :PRINUM ; PRINT LINE #

   * * *
Source Code

B992: ERRDONE
B992 206EBD JSR PCCR ; PRINT CR
B995 A908 LDA #$0 ; CLEAR A
B997 85B9 STA ERRNUM ; CLEAR ERROR #
B999 4C6A8 JMP SYNTAX

Print Integer Number in FR0

B99C :PRINUM
B99C 20AAD9 JSR CVIFF ; CONVERT TO FLOATING POINT
B99F 20E6D8 JSR CVFASC ; CONVERT TO ASCII
B9A2 A5F3 LDA INBUFF ; GET ADR OF #$ LOW
B9A4 8595 STA SRCADR ; SET FOR PRINT ROUTINE
B9A6 A5F4 LDA INBUFF+1 ; GET ADR OF #$ HIGH
B9A8 8596 STA SRCADR+1 ; SET FOR PRINT ROUTINE
B9AA 2035B5 JSR LPRTOKEN ; GO PRINT ERROR #
B9AD 60 RTS

B9AE 204154204C 'ERMS DC 'AT LINE'
494E45A0

Execute Graphics Routines

XSETCOLOR — Execute SET COLOR

B9B7 XSETCOLOR
B9B7 20E9AB JSR GETINT ; GET REGISTER #
B9BA A5D4 LDA FR0 ; GET #
B9BC 905 CMP #$5 ; IS IT <5?
B9BE B01A 'B9DA BCS :ERCOL ; IF NOT, ERROR
B9C0 48 PHA ; SAVE
B9C1 20E9AB JSR GETINT ; GET VALUE
B9C4 A5D4 LDA FR0 ; GET VALUE*16+6
B9C6 ASLA A ; X
B9C7 +0A ASL A ; X
B9C7 +0A ASL A ; X
B9CB ASLA A ; X
B9C8 +0A ASLA A ; X
B9C9 +0A ASLA A ; X
B9CA 48 PHA ; SAVE ON STACKS
B9CB 20E9AB JSR GETINT ; GET VALUE
B9CE 68 PLA ; GET VALUE 2*16 FROM STACK
B9CF 10 CLC
B9D0 65D4 ADC FR0 ; ADD IN VALUE 3
B9D2 A8 TAY ; SAVE VALUE 2*16 + VALUE 5
B9D3 68 PLA ; GET INDEX
B9D4 AA TAX ; PUT IN X
B9D5 90 TYA ; GET VALUE
B9D6 9DCA02 STA CREGS,X ;SET VALUE IN REGS
B9D9 60 RTS

B9DA :ERSND
B9DA :ERCOL
B9DA 203AB9 JSR ERVAL

XSOUND — Execute SOUND

B9DD XSOUND
B9DD 20E9AB JSR GETINT ; GET 1 BYTE INTEGER
B9E0 A5D4 LDA FR0 ; X
B9E2 C904 CMP #$4 ; IS IT <47
B9E4 B0F4 'B9DA BCS :ERSND ; IF NOT, ERROR
Source Code

; Source Code

B9E6 ASLA A ; GET VALUE *2
B9E6 +0A ASL ;
B9E7 4B PHA ;

B9E8 A900 LDA #0 ; SET TO ZERO
B9EA 800BD2 STA SREG1 ; X

B9ED A903 LDA #3
B9EF 800FD2 STA SKCTL ;

B9F2 20E90B JSR GETINT ; GET EXP2
B9F5 6B PLA ; GET INDEX
B9F6 4B PHA ; SAVE AGAIN
B9F7 AA TAX ; PUT IN INDEX REG
B9F8 A5D4 LDA FR0 ; GET VALUE
B9FA 900BD2 STA SREG2,X ; SAVE IT

B9FD 20E90B JSR GETINT ; GET EXP3
BA00 A5D4 LDA FR0 ; GET 16*EXP3 ; X
BA02 ASLA ;
BA02 +0A ASL A ; X
BA03 ASLA ;
BA03 +0A ASL A ; X
BA04 ASLA ;
BA04 +0A ASL A ; X
BA05 ASLA ;
BA05 +0A ASL A ; X
BA06 4B PHA ; SAVE IT

BA07 20E90B JSR GETINT ; GET EXP4
BA0A 6B PLA ; GET 16*EXP3
BA0B A8 TAY ; SAVE IT
BA0C 6B PLA ; GET INDEX
BA0D AA TAX ; PUT IN X
BA0E 98 TYA ; GET EXP3*16
BA0F 18 CLC
BA10 65D4 ADC FR0 ; GET 16*EXP3+EXP4
BA12 9001D2 STA SREG3,X ; STORE IT
BA15 60 RTS

XPOS — Execute POSITION

BA16 XPOS
BA16 20E90B JSR GETINT ; GET INTEGER INTO FR0
BA19 A5D4 LDA FR0 ; SET X VALUE
BA1B 8555 STA SCRX ; X
BA1D A5D5 LDA FR0+1 ; X
BA1F 8556 STA SCRX+1 ; X

BA21 20E90B JSR GET1INT ; SET Y VALUE
BA24 A5D4 LDA FR0 ; X
BA26 8554 STA SCRY ; X
BA28 60 RTS

XCOLOR — Execute COLOR

BA29 XCOLOR
BA29 20E90B JSR GETINT ; GET INTEGER INTO FR0
BA2C A5D4 LDA FR0 ;
BA2E 85C8 STA COLOR
BA30 60 RTS

XDRAWTO — Execute DRAWTO

BA31 XDRAWTO
BA31 2016BA JSR XPOS ; GET X,Y POSITION
BA34 A5C8 LDA COLOR ; GET COLOR
BA36 8DFB02 STA SVCOLOR ; SET IT

233
Source Code

BA39 A911 LDA #ICDRAW ; GET COMMAND
BA3B A206 LDX #6 ; SET DEVICE
BA3D 20C4BA JSR GLPCX ; SET THEM
; BA40 A90C LDA #$6C ; SET AUX 1
BA42 9D4AA3 STA ICAUX1,X
BA45 A900 LDA #0 ; SET AUX 2
BA47 9D4BB3 STA ICAUX2,X
BA4A 2024BD JSR IO7
BA4D 4CB3BC JMP IOTEST

XGR — Execute GRAPHICS

BA50 A206 LDX #6 ; GET DEVICE
BA52 86C1 STX IODVC ;SAVE DEVICE #
BA54 20F1BC JSR CLSYS1 ; GO CLOSE IT
BA57 20E0AB JSR GETINT ; GET INTEGER INTO FR0
; BA5A A273 LDX #$SSTR&255 ; SET INBUFF TO POINT
BA5C A0B3 LDY #$SSTR/256 ; TO FILE SPEC STRING
BA5E 86F3 STX INBUFF ; X
BA60 84F4 STY INBUFF+1 ; X
; BA62 A206 LDX #6 ; GET DEVICE #
BA64 A5D4 LDA FR0 ;SET SOME BITS FOR GRAPHICS
BA66 29F0 AND #$F0 ;
BA68 491C EOR #$1CGR ;
BA6A A8 TAY ;
BA6B A5D4 LDA FR0 ; GET AUX2 [GRAPHICS TYPE]
BA6D 2011BB JSR SOPEN ; OPEN
BA70 4CB3BC JMP IOTEST ; TEST I/O OK
; BA73 533A9B SSTR DB 'S:',CR

XPLT — Execute PLOT

BA76 2016BA JSR XPOS ; SET X,Y POSITION
; BA79 A5C8 LDA COLOR ; GET COLOR
BA7B A206 LDX #6 ; GET DEVICE #
BA7D 4CA1BA JMP PRCX ; GO PRINT IT

Input/Output Routines

BA80 LOCAL

GETLINE — Get a Line of Input

; GLINE - GET LINE [PROMPT ONLY]
; GNLINE - GET NEW LINE [CR, PROMPT]
; BA80 GNLINE
BA80 A6B4 LDX ENTDTD ; IF ENTER DEVICE NOT ZERO
BA82 D00E 'BA92 BNE GLGO ; THEN DO PROMPT
BA84 A99B LDA #CR ; PUT EOL
BA86 209FBA JSR PUTCHAR
; BA89 GLINE
BA89 A6B4 LDX ENTDTD ; IF ENTER DEVICE NOT ZERO
BA8B D005 'BA92 BNE GLGO ; THEN DON'T PROMPT
BA8D A5C2 LDA PROMPT ; PUT PROMPT
BA8F 209FBA JSR PUTCHAR
; BA92 GLGO
BA92 A6B4 LDX ENTDTD
BA94 A985 LDA #ICGTR
Source Code

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>BA96</td>
<td>JSR GLPCX</td>
<td>; GO DO I/O</td>
</tr>
<tr>
<td>BA99</td>
<td>JSR I01</td>
<td>; GO TEST RESULT</td>
</tr>
<tr>
<td>BABC</td>
<td>JMP IOTEST</td>
<td>;</td>
</tr>
</tbody>
</table>

**PUTCHAR — Put One Character to List Device**

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>BAF9</td>
<td>PRCX</td>
<td>;</td>
</tr>
<tr>
<td>BAF9</td>
<td>JSR PUTCHE</td>
<td>;</td>
</tr>
<tr>
<td>BAF9</td>
<td>A685</td>
<td>; GET LIST DEVICE</td>
</tr>
<tr>
<td>BAAS</td>
<td>LDX LISTDTD</td>
<td>;</td>
</tr>
<tr>
<td>BAAS</td>
<td>PRCX</td>
<td>;</td>
</tr>
<tr>
<td>BAAS</td>
<td>JSR GLPX</td>
<td>; SET DEVICE</td>
</tr>
<tr>
<td>BABA</td>
<td>LDA ICAUX1,X</td>
<td>; SET UP ZERO PAGE IOCB</td>
</tr>
<tr>
<td>BABA</td>
<td>STA ICAUX1-IOCB+ZICB</td>
<td>;</td>
</tr>
<tr>
<td>BABA</td>
<td>LDA ICAUX2,X</td>
<td>;</td>
</tr>
<tr>
<td>BABA</td>
<td>STA ICAUX2-IOCB+ZICB</td>
<td>;</td>
</tr>
<tr>
<td>BABA</td>
<td>PLA</td>
<td>;</td>
</tr>
<tr>
<td>BABA</td>
<td>TAY</td>
<td>;</td>
</tr>
<tr>
<td>BABA</td>
<td>JSR PDUM</td>
<td>;</td>
</tr>
<tr>
<td>BAB4</td>
<td>TAY</td>
<td>; RETURN HERE FROM ROUTINE</td>
</tr>
<tr>
<td>BAB5</td>
<td>JSR IOTES2</td>
<td>; TEST STATUS</td>
</tr>
<tr>
<td>BAB8</td>
<td>PRCX</td>
<td>;</td>
</tr>
<tr>
<td>BAB8</td>
<td>JSR PDUM</td>
<td>;</td>
</tr>
<tr>
<td>BAB8</td>
<td>BD4783</td>
<td>; GO TO PUT ROUTINE</td>
</tr>
<tr>
<td>BAB8</td>
<td>PHA</td>
<td>;</td>
</tr>
<tr>
<td>BAB8</td>
<td>LDA ICPUT+1,X</td>
<td>;</td>
</tr>
<tr>
<td>BAB8</td>
<td>PHA</td>
<td>;</td>
</tr>
<tr>
<td>BAB8</td>
<td>LDA ICPUT,X</td>
<td>;</td>
</tr>
<tr>
<td>BAB8</td>
<td>PLA</td>
<td>;</td>
</tr>
<tr>
<td>BAB8</td>
<td>TAY</td>
<td>;</td>
</tr>
<tr>
<td>BAB8</td>
<td>LDA #$92</td>
<td>; LOAD VALUE FOR CIO ROUTINE</td>
</tr>
<tr>
<td>BAB8</td>
<td>STA IOCMD</td>
<td>;</td>
</tr>
<tr>
<td>BAB8</td>
<td>LDDVX</td>
<td>;</td>
</tr>
<tr>
<td>BAB8</td>
<td>#$04</td>
<td>; OPEN INPUT</td>
</tr>
<tr>
<td>BAB8</td>
<td>JSR ELADVC</td>
<td>; GO OPEN ALT DEVICE</td>
</tr>
<tr>
<td>BAD0</td>
<td>STA LSTDTD</td>
<td>; SET ENTER DEVICE</td>
</tr>
<tr>
<td>BAD2</td>
<td>JMP SYNTAX</td>
<td>;</td>
</tr>
<tr>
<td>BAC9</td>
<td>4C60A0</td>
<td>;</td>
</tr>
<tr>
<td>BAC9</td>
<td>JSR GLPCX</td>
<td>;</td>
</tr>
<tr>
<td>BAC6</td>
<td>STA IOCMD</td>
<td>; AS I/O DEVICE</td>
</tr>
<tr>
<td>BAC9</td>
<td>86C1</td>
<td>;</td>
</tr>
<tr>
<td>BAC9</td>
<td>STX IOTEST</td>
<td>;</td>
</tr>
<tr>
<td>BAC9</td>
<td>JSR 2DDBA</td>
<td>;</td>
</tr>
<tr>
<td>BACB</td>
<td>LDA #$04</td>
<td>; OPEN INPUT</td>
</tr>
<tr>
<td>BACD</td>
<td>JSR ELADVC</td>
<td>; GO OPEN ALT DEVICE</td>
</tr>
<tr>
<td>BAD0</td>
<td>STA LSTDTD</td>
<td>; SET LIST DEVICE</td>
</tr>
<tr>
<td>BAD2</td>
<td>JMP SYNTAX</td>
<td>;</td>
</tr>
</tbody>
</table>

**XENTER — Execute ENTER**

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>BACB</td>
<td>XENTER</td>
<td>;</td>
</tr>
<tr>
<td>BACB</td>
<td>A904</td>
<td>;</td>
</tr>
<tr>
<td>BACD</td>
<td>JSR ELADVC</td>
<td>;</td>
</tr>
<tr>
<td>BAD0</td>
<td>STA LSTDTD</td>
<td>;</td>
</tr>
<tr>
<td>BAD2</td>
<td>JMP SYNTAX</td>
<td>;</td>
</tr>
</tbody>
</table>

**FLIST — Open LIST File**

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>BAD5</td>
<td>FLIST</td>
<td>;</td>
</tr>
<tr>
<td>BAD5</td>
<td>LDA #$08</td>
<td>; OPEN OUTPUT</td>
</tr>
<tr>
<td>BAD7</td>
<td>JSR ELADVC</td>
<td>; GO OPEN ALT DEVICE</td>
</tr>
<tr>
<td>BAD4</td>
<td>STA LSTDTD</td>
<td>; SET LIST DEVICE</td>
</tr>
<tr>
<td>BADC</td>
<td>60</td>
<td>; DONE</td>
</tr>
<tr>
<td>BADD</td>
<td></td>
<td>;</td>
</tr>
<tr>
<td>BADD</td>
<td>48</td>
<td>;</td>
</tr>
<tr>
<td>BAD2</td>
<td>LDA #$7</td>
<td>; USE DEVICE 7</td>
</tr>
<tr>
<td>BAE0</td>
<td>STA IODVC</td>
<td>; SET DEVICE</td>
</tr>
<tr>
<td>BAE0</td>
<td>LDA #$108</td>
<td>; BEFORE</td>
</tr>
<tr>
<td>BAE5</td>
<td>JSR LDDVX</td>
<td>; BEFORE</td>
</tr>
<tr>
<td>BAE7</td>
<td>LDA #$ICCLOSE</td>
<td>; GO CLOSE DEVICE</td>
</tr>
<tr>
<td>BAE7</td>
<td>JSR 108</td>
<td>; OPEN OF NEW ONE</td>
</tr>
<tr>
<td>BAE8</td>
<td>LDA #$ICOIO</td>
<td>; CMD IS OPEN</td>
</tr>
<tr>
<td>BAE8</td>
<td>STY IOCMD</td>
<td>;</td>
</tr>
<tr>
<td>BAE8</td>
<td>PLA</td>
<td>;</td>
</tr>
<tr>
<td>BAF0</td>
<td>LDA #$0</td>
<td>; GET AUX2</td>
</tr>
<tr>
<td>BAF1</td>
<td>JSR XOP2</td>
<td>; GO OPEN</td>
</tr>
</tbody>
</table>
Source Code

RUN from File

BAF7 A9FF  FRUN  LDA  #$FF  ; SET RUN MODE

XLOAD — Execute LOAD Command

BAFB  XLOAD  LDA  #$0  ; SET LOAD MODE
BAFD  48  :LD0  PHA  ; SAVE R/L TYPE
BAFE  A904  LDA  #$04  ; GO OPEN FOR INPUT
BB00  20DBBA  JSR  ELADVC  ; THE SPECIFIED DEVICE
BB03  68  PLA  ; GET R/L TYPE
BB04  XLOAD1  XLOAD  LDA  #$1CGTC  ; CMD IS GET TEXT CHARS
BB05  A907  STA  IOCMD  ; SET LOAD IN PROGRESS
BB09  B5CA  STA  LOADFLG
BB0B  20A6BC  JSR  LDDVX  ; LOAD DEVICE X REG
BB0E  A90E  LDY  #ENDSTAR-OUTBUFF  ; Y = REC LENGTH
BB10  2010BD  JSR  IO3  ; GO GET TABLE BLOCK
BB13  20B3BC  JSR  IOTEST  ; TEST I/O
BB16  A80805  LDA  MISCRAM+OUTBUFF  ; IF FIRST 2
BB19  B80105  ORA  MISCRAM+OUTBUFF+1  ; BYTES NOT ZERO
BB1C  D038 "BB56  BNE  :LDFER  ; THEN NOT SAVE FILE
BB1E  A28C  LDX  #STARP  ; START AT STARP DISPL
BB20  10  :LD1  CLC
BB21  A580  LDA  OUTBUFF  ; ADD LOMEM TO
BB23  7D0005  ADC  MISCRAM,X  ; LOAD TABLE DISPL
BB26  A8  TAY
BB27  A581  LDA  OUTBUFF+1
BB29  7D0105  ADC  MISCRAM+1,X
BB2C  CDE062  CMP  HIMEM+1  ; IF NEW VALUE NOT
BB2F  900A "BB3B  BCC  :LD3  ; LESS THEN HIMEM
BB31  D003 "BB3B  BNE  :LD2  ; THEN ERROR
BB33  CCE052  CPY  HIMEM  ; IF NOT AT LOWER ENTRY
BB36  9003 "BB3B  BCC  :LD3  ; CONTINUE
BB38  4C1AB9  :LD2  JMP  ERRPTL
BB3B  9501  :LD3  STA  1,X  ; ELSE SET NEW TABLE VALUE
BB3D  9400  STY  0,X  ; DECREMENT TO PREVIOUS TBL
BB3F  CA  DEX
BB40  CA  DEX
BB41  E0B2  CPX  #VNTP  ; IF NOT AT LOWER ENTRY
BB43  B0DB "BB20  BCS  :LD1  ; THEN CONTINUE
BB45  2088BB  JSR  :LSBLK  ; LOAD USER AREA
BB48  2066B7  JSR  XCLR  ; EXECUTE CLEAR
BB4B  A900  LDA  #$0  ; RESET LOAD IN PROGRESS
BB4D  B5CA  STA  LOADFLG
BB4F  68  PLA
BB50  F001 "BB53  BEQ  :LD4  ; BR IF LOAD
BB52  60  RTS
BB53  4C58A6  JMP  SNX1  ;GO TO SYNTAX
BB56  :LDPER
BB5A  A900  LDA  #$0  ; RESET LOAD IN PROGRESS
BB5B  B5CA  STA  LOADFLG
BB5F  2016B9  JSR  ERRNSF  ; NOT SAVE FILE
Source Code

**XSAVE — Execute SAVE Command**

BB5D  XSAVE
BB5D  A908  LDA  #08  ;  GO OPEN FOR OUTPUT
BB5F  20DDBA  JSR  ELADVC  ;  THE SPECIFIED DEVICE
BB62  XSAVE1
BB62  A908  LDA  #1CPTC  ;  I/O CMD IS PUT TEXT CHARS
BB64  85C0  STA  IOCMD  ;  SET I/O CMD
BB66  A280  LDX  #OUTBUFF  ;  MOVE RAM TABLE PTRS
BB68  38  :SV1  SEC
BB69  B500  LDA  0,X  ;  TO LBUFF
BB6B  E580  SBC  OUTBUFF  ;  AS DISPLACEMENT
BB6D  900005  STA  MISCRA,M,X  ;  FROM LOW MEM
BB70  E8  INX
BB71  B500  LDA  0,X
BB73  E581  SBC  OUTBUFF+1
BB75  900005  STA  MISCRA,M,X
BB78  E8  INX
BB79  E00E  CPX  #ENDSTAR
BB7B  90EB  *BB68  BCC  :SV1
BB7D  20A6BC  JSR  LDDVX  ;  OUTPUT LBUFF
BB80  A00E  LDY  #ENDSTAR-OUTBUFF  ;  FOR PROPER LENGTH
BB82  2010BD  JSR  IO3
BB85  2030BC  JSR  IOTEST  ;  TEST GOOD I/O

**LSBLK — LOAD or SAVE User Area as a Block**

BB88  :LSBLK
BB88  20A6BC  JSR  LDDVX  ;  LOAD DEVICE X REG
BB8B  A582  LDA  VNTP  ;  SET VAR NAME TBL PTR
BB8D  85F3  STA  INBUFF  ;  AS START OF BLOCK ADR
BB8F  A583  LDA  VNTP+1
BB91  85F4  STA  INBUFF+1
BB93  AC8005  LDY  MISCRA+STARP+1  ;  A,Y = BLOCK LENGTH
BB96  8E  DEY
BB97  9B  TYA
BB98  AC9C05  LDY  MISCRA+STARP
BB9B  2012BD  JSR  IO4  ;  GO DO BLOCK I/O
BB9E  2030BC  JSR  IOTEST
BBA1  4C1FBC  JMP  CLSYS1  ;  DO SAVE

**XCSAVE — Execute CSAVE**

BBAC  XCSAVE
BBAC  A908  LDA  #8  ;  GET OPEN FOR OUTPUT
BBA6  20B6BB  JSR  COPEN  ;  OPEN CASSETTE
BBA9  4C62BB  JMP  XSAVE1  ;  DO SAVE

**XCLOAD — Execute CLOAD**

BBAC  XCLOAD
BBAC  A904  LDA  #4  ;  GET OPEN FOR OUTPUT
BBAE  20B4BB  JSR  COPEN  ;  OPEN CASSETTE
BBB1  A900  LDA  #0  ;  GET LOAD TYPE
BBB3  4C04BB  JMP  XLOAD1  ;  DO LOAD

**COPEN — OPEN Cassette**

* ON ENTRY:  A - TYPE OF OPEN [IN OR OUT]
* ON EXIT:  A - DEVICE #7

BBB6  COPEN
BBB6  48  PHA
BBB7  A2CE  LDX  #:CSTR&255
BBB9  B6F3  STX  INBUFF

237
**Source Code**

```
BBBB A2BB  LDX  #:CSTR/256
BBBD 86P4  STX  INBUFF+1
;
BBBF A207  LDX  $7
BBC1 6B   PLA
BBC2 A8   TAY
BBC3 A9B0 LDA  #$80
;
BBBC 2B01BB JSR  SOPEN
BBBC 2B03BC JSR  IOTEST
BBBC A907 LDA  $7
BBBC 6B   RTS
;
BBCE 433A9B :CSTR  DB  'C:',CR

**SOPEN — OPEN System Device**

* ON ENTRY  X - DEVICE
*         Y - AUX1
*         A - AUX2
*         INBUFF - POINTS TO FILE SPEC

BBBD 4B   PHA
BBBD A003 LDA  #ICOIO
BBBD 2B0C48A JSR  GPCX
BBBD 6B   PLA
BBBD 9B4B03 STA  ICAUX2,X
BBBD 9B   TYA
BBBD 9B4B03 STA  ICAUX1,X
;
BBBF 2B19BB JSR  IO5
BBBE 4C51DA JMP  INTLBF

**XXIO — Execute XIO Statement**

BBE5 2B04BD JSR  GIOCMD
BBEB 4CEDBB JMP  XOP1

**XOPEN — Execute OPEN Statement**

BBEB A003 LDA  #ICOIO
BBEB 85C0 XOP1 STA  IOCMD
BBEF 2B09BC JSR  GIOVC
BBF2 2B04BD JSR  GIOCMD
BBF5 4B   PHA
BBF6 2B04BD JSR  GIOCMD
BBF9 A8   TAY
BBFA 6B   PLA
BBFB 4B   XOP2
BBFB 48   PHA
BBFC 9B   TYA
BBFD 4B   PHA
BBFE 2B0E0A JSR  EXEXPR
BC01 2B79BD JSR  SETSEOL
BC04 2B06BC JSR  LDDVX
BC07 6B   PLA
BC08 9B4B03 STA  ICAUX2,X
BC0B 6B   PLA
BC0C 9B4B03 STA  ICAUX1,X
BC0F 2B00BD JSR  IO1
BC12 2B99BD JSR  RSTSEOL
```

XXIO — Execute XIO Statement

XOPEN — Execute OPEN Statement
Source Code

BC15 2851DA JSR INTLBF ; GO TEST I/O STATUS
BC18 4CB3BC JMP IOTEST

XCLOSE — Execute CLOSE
BC1B XCLOSE
BC1B A90C LDA #ICCLOSE ; CLOSE CMD

GDVCIO — General Device I/O
BC1D GDVCIO
BC1D 85C0 STA IOCMD ; SET CMD
BC1F 289FBC JSR GIODVC ; GET DEVICE
BC22 2824BD GDI01 JSR I07 ; GO DO I/O
BC25 4CB3BC JMP IOTEST ; GO TEST STATUS

XSTATUS — Execute STATUS
BC28 XSTATUS
BC28 289FBC JSR GIODVC ; GET DEVICE
BC2B A90D LDA #$26 ; NOTE CMD
BC2C 2826BC JSR I08 ; GO GET STATUS
BC30 28FBC JSR LDIOSTA ; LOAD STATUS
BC33 4C2DBD JMP ISVAR1 ; GO SET VAR

XNOTE — Execute NOTE
BC36 XNOTE
BC36 A926 LDA #$26 ; NOTE CMD
BC39 281DBC JSR GDVCIO ; GO DO
BC3B BD4C03 LDA ICAUX3,X ; GET SECTOR N/. LOW
BC3E BC4D03 LDY ICAUX4,X ; AND HI
BC41 282FBD JSR ISVAR ; GO SET VAR
BC44 28A6BC JSR LDDVX ; GET DEVICE X REG
BC47 BD4E03 LDA ICAUX5,X ; GET DATA LENGTH
BC4A 4C2DBD JMP ISVAR1 ; GO SET VAR

XPOINT — Execute POINT
BC4D XPOINT
BC4D 289FBC JSR GIODVC ; GET I/O DEVICE NO.
BC50 28D5AB JSR GETPRINT ; GET SECTOR NO.
BC53 28A6BC JSR LDDVX ; GET DEVICE X
BC56 28D4 LDA PR0 ; SET SECTOR NO.
BC5B 9D4C03 STA ICAUX3,X
BC5B A5D4 LDA PR0+1
BC5D BD4D03 STA ICAUX4,X
BC60 28D5AB JSR GETPRINT ; GET DATA LENGTH
BC63 28A6BC JSR LDDVX ; LOAD DEVICE X
BC66 28D4 LDA PR0 ; GET AL
BC66 9D4C03 STA ICAUX5,X ; SET DATA LENGTH
BC6A 28D4 STA #$25 ; SET POINT CMD
BC6C 85C0 STA IOCMD
BC6F 4C22BC JMP GDI01 ; GO DO

XPUT — Execute PUT
BC72 XPUT
BC72 289FBC JSR GIODVC ; GET DEVICE #
BC75 28E8AB JSR GETINT ; GET DATA
BC7B A5D4 LDA PR0 ; X
BC7A 4CA1BA JMP IOVC

XGET — Execute GET
BC7F XGET
BC7F 289FBC JSR GIODVC ; GET DEVICE
BC81 GET1
BC82 BD4C03 LDA #ICGYC ; GET COMMAND
BC84 85C0 STA IOCMD ; SET COMMAND

239
BC86  A001  LDY  #1  ; SET BUFF LENGTH=1
BC88  2010BD  JSR  IO3  ; DO IO
BC8B  20B3BC  JSR  IOTEST  ; TEST I/O
BC98  A000  LDY  #0  ; GET CHAR
BC9B  B1F3  LDA  [INBUFF],Y  ; X
BC92  4C2BB  JMP  ISVAR1  ; ASSIGN VAR

XLOCATE — Execute LOCATE
BC95  2016BA  JSR  XPOS  ; GET X,Y POSITION
BC98  A206  LDX  #6  ; GET DEVICE #
BC9A  20C6BA  JSR  GLPX  ; X
BC9D  D0E3  BNE  GET1  ; GO GET

GIODVC — Get I/O Device Number
BC9F  GIODVC
BCA2  85C1  STA  IODVC  ; SET AS DEVICE
BCAF  F00A  ~BCB0  BEQ  DNERR  ; BR IF DVC=0

LLDVX — Load X Register with I/O Device Offset
B6A6  A5C1  LDA  IODVC  ; GET DEVICE
B6A8  ASLA  ASL  A  ; MULT BY 16
B6A9  +0A  ASLA  A
B6AA  +0A  ASLA  A
B6AB  +0A  ASLA  A
B6AC  +0A  ASLA  A
B6AD  AA  TAX  ; PUT INTO X
B6AE  3001  ~BCB0  BMI  DNERR  ; BR DIN0>7
B6AF  60  RTS  ; AND RETURN
B6B0  2018B9  DNERR  JSR  ERRDNO

IOTEST — Test I/O Status
B6B3  IOTEST
B6B5  20FBBC  JSR  LDIOSTA  ; LOAD I/O STATUS
B6B6  IOTEST2
B6B8  60  RTS  ; ELSE RETURN
B6B9  SICKIO
B6BA  A000  LDY  #0  ; RESET DISPLAY FLAG
B6BB  8CPE02  STY  DSPPLG
;
B6BE  C980  CMP  #$C8BRK  ; IF BREAK
B6C0  D00A  ~BCCC  BNE  :SIO1  ; SIMULATE ASYNC
B6C2  8411  STY  BRKBYT  ; BREAK
B6C4  A5CA  LDA  LOADFLG  ; IF LOAD FLAG SET
B6C6  F003  ~BCCB  BEQ  :SIOS  ;
B6C8  4C00A0  JMP  COLDSTART  ;DO COLDSTART
B6CB  :SIOS
B6CC  60  RTS  ;
;
B6CC  A4C1  :SIO1  LDY  IODVC  ; PRE-LOAD I/O DEVICE
B6CE  C998  CMP  #$98  ; WAS ERROR EOF
B6CD  F00F  ~BCE1  BEQ  :SIO4  ; BR IF EOF
B6D2  85B9  :SIO2  STA  ERRNUM  ; SET ERROR NUMBER
;
B6D4  C907  CPY  #7  ; WAS THIS DEVICE #7
B6D6  D003  ~BCDB  BNE  :SIO3  ; BR IF NOT
B6D8  20F1BC  JSR  CLSYSD  ; CLOSE DEVICE 7
;
B6DB  2072BD  :SIO3  JSR  SETDZ  ; SET L/D DEVICE = 0
B6DE  4C40B9  JMP  ERROR  ; REPORT ERROR

240
BCE1 C007 :ST04 CPY #7 ; WAS EOF ON DEVICE 7
BCE3 D00D 9CD2 BNE :ST02 ; BR IF NOT
BCE5 A25D LDX #EPCHAR ; WERE WE IN ENTER
BCE7 E402 CPX PROMPT ;
BCE9 D007 9CD2 BNE :ST02 ; BR NOT ENTER
BCEB 2001BC JSR CLSYSD ; CLOSE DEVICE 7
BCEE 4C53A0 JMP SNX2 ; GO TO SYNTAX

CLSYSD — Close System Device
BCF1 CLSYSD
BCF1 20A6BC CLSYS1 JSR LDDVX
BCF4 F00B 2D01 BEQ NOCD0 ; DON'T CLOSE DEVICES
BCF6 A90C LDA #ICCLOSE ; LOAD CLOSE CORD
BCF8 4C26BD JMP 108 ; GO CLOSE

LDIOSTA — Load I/O Status
BCFB LDIOSTA
BCFB 20A6BC JSR LDDVX ; GET DEVICE X REG
BCFE BD4303 LDA ICSTA,X ; GET STATUS
BD01 60 RTS
BD01 60 CLO l

GIOPRM — Get I/O Parameters
BD02 GIOPRM
BD02 E6A8 INC STINDEX ; SKIP OVER #
BD04 20D5AB GIOCMD JSR GETPINT ; GET POSITIVE INT
BD07 A5D4 LDA FR0 ; MOVE LOW BYTE TO
BD09 60 RTS

I/O Call Routine
BD0A A0FF IO1 LDY #255 ; BUFL = 255
BD0C D002 "BD10 BNE IO3 ; BUFL = 0
BD0E A000 IO2 LDA #0 ; BUFL < 256
BD10 A900 IO3 LDA #0 ; SET BUFL
BD12 9D4903 IO4 STA ICBH,X
BD15 98 TYA
BD16 9D4803 STA ICBL,X
BD19 A5F4 IO5 LDA INBUFF+1 ; LOAD INBUFF VALUE
BD1B A4F3 LDI INBUFF ; SE BUF ADR
BD1D 9D4503 IO6 STA ICBA,X
BD20 98 TYA
BD21 9D4403 STA ICBA,X
BD24 A5C0 IO7 LDA IOCMD ; LOAD COMMAND
BD26 9D4203 IO8 STA ICCOM,X ; SET COMMAND
BD29 2056E4 JSR C10 ; GO DO I/O
BD2C 60 RTS ; DONE

ISVAR — I/O Variable Set
BD2D A000 ISVAR ISVAR1
BD2D A000 LDY #0 ; GET HIGH ORDER BYTE
BD2F ISVAR
BD2F 48 PHA ; PUSH INT VALUE LOW
BD30 98 TYA
BD31 48 PHA
BD32 200FAC JSR POP1
BD35 68 PLA ; PUSH INT VALUE HI
BD36 85D5 STA FR0+1 ; GET VARIABLE
BD38 68 PLA
BD39 85D4 STA FR0 ; CONVERT TO FP
BD3B 20AAD9 JSR CVIFP
BD3E 4C16AC JMP RTNVAR ; AND RETURN TO TABLE

241
CLSALL — CLOSE All IOC Bs [except 0]

BD41    CLSALL
         ;
         ; TURN OFF SOUND
         ;
   BD41  A900  LDA  #0
   BD43  A207  LDX  #7
   BD45  :CL
   BD45  9D00D2  STA  SREG3-1,X
   BD48  CA  DEX
   BD49  DBFA  "BD45  BNE  :CL
   BD4B  A087  LDY  #7
   BD4D  B4C1  STY  IODVC
   BD4F  28F1BC  CLALL  JSR  28F1CLSYSD
   BD52  C6CL  DEC  IODVC
   BD54  D8F9  "BD4F  BNE  CLALL1
   BD56  60  RTS

PREADY — Print READY Message

BD57  PREADY
   BD57  A206  LDX  #RML-1
   BD59  B6F2  PRDY1  STX  CIX
   BD5B  BD67BD  LDA  RMSG,X
   BD5E  289FBA  JSR  PRCHAR
   BD61  A6F2  LDX  CIX
   BD63  CA  DEX
   BD64  18F3  "BD59  BPL  PRDY1
   BD66  60  RTS
   BD67  9B59444145  RMSG  DB  CR, 'YDAER', CR
       529B  = 0007  RML  EQU  *-RMSG

PRCR — Print Carriage Return

BD68  A300  PRCR  LDX  #0
   BD6B  F6E7  "BD59  BEQ  PRDY1

SETDZ — Set Device 0 as LIST/ENTER Device

BD72  A900  SETDZ  LDA  #0
   BD74  85B4  STA  ENTDTD
   BD76  85B5  STA  LSTTDTD
   BD78  60  RTS

SETSEOL — Set an EOL [Temporarily] after a String EOL

BD79  SETSEOL
   BD79  289B2B  JSR  AAPSTR
   BD7C  A5D4  LDA  FR8-2+EVSADR
   BD7E  85F3  STA  INBUFF
   BD80  85F3  LDA  FR8-1+EVSADR
   BD82  85F4  STA  INBUFF+1

   BD84  A4D6  LDY  FR8-2+EVSLEN
   BD86  A6D7  LDX  FR8-1+EVSLEN
   BD88  F002  "BD8C  BEQ  #$FF
   BD8A  A9FF  LDY  #$FF
   BD8C  B1F3  :SSE1  LDA  [INBUFF],Y
   BD8E  8597  STA  INDEX2
   BD90  B498  STY  INDEX2+1
   BD92  A99B  LDA  #CR
   BD94  91F3  STA  [INBUFF],Y
   BD96  8592  STA  MEOLFLG
   BD98  60  RTS
   BD99  RSTSEOL
   BD99  A498  LDY  INDEX2+1

242
Source Code

```
B09B A597 LDA INDEX2 ; LOAD CHAR
B09D 91F3 STA [INBUFF], Y ; DONE
B09F A900 LDA #0 ;
BDA1 8592 STA MEOLFLG ; RESET EOL FLAG
BDA3 68 RTS ; DONE
BDA4 = 0001 PATCH DS PATSZ

SIN[X] and COS[X]

BDA5 38 SINERR SEC ; ERROR - SET CARRY
BDA6 60 RTS

BDA7 A904 SIN LDA #4 ; FLAG SIN[X] ENTRY RIGHT NOW
BDA9 24D4 BIT FR0
BDBA 1BBE BPL BOTH
BDBD A902 LDA #2 ; SIN[-X]
BDAF D002 BDB 3 BNE BOTH

BDB3 85F0 BOTH STA SGNFLG
BDB5 A5D4 LDA FR0 \#$7F ; FORCE POSITIVE
BDB7 297F AND \#$7F
BDB9 85D4 STA FR0

BDBB A95F LDA #PIOV2&$FF
BDBD 65FB ADC DEGFLG

BDC0 AA TAX
BDC1 A0BE LDY #PIOV2/$100
BDC3 289DD JSR FLD1R
BDC6 282DD JSR FDIV X/[PI/2] OR X/90

BDC8 9001 ^BDDC BCC SINF7
BDB8 60 SINOVF RTS ; OVERFLOW
BDB9 C545 SINF7

BDBD E940 SBC #$40
BDD3 38 SEC

BDD5 C904 CMP #FPREC-2 FIND QUAD NO & REMAINDER
BDD7 18CC ^BDA5 BPL SINERR ; OUT OF RANGE
BDDD AA TAX
BDDA 85DF LDA FRB+1,X ; LSB
BDDC 85FI STA XFMFLG
BDE0 2910 AND \#$10 ; CHECK 10'S DIGIT
BDE2 A902 LDA #2 ; ODD - ADD 2 TO QUAD #

BDE4 18 SINF5 CLC
BDE5 65FI ADC XFMFLG
BDE7 2903 AND #3 ; QUADRANT = 0,1,2,3
BDE9 65F0 ADC SGNFLG ; ADJUST FOR SINE VS COSINE
BDEB 85F0 STA SGNFLG

BDEF 286DD JSR PMOVE

BDF1 A6F1 LDX XFMFLG
BDF4 A900 LDA \#0
BDF6 95E2 SINF1 STA PR1+2,X ; CLEAR FRACTION
BDF8 E8 INX
BDFB 28 INX
BDF9 E003 CPX #FPREC-3
BDFB 90F9 ^BDF6 BCC SINF1
BDFD 286DD JSR PSUB ; LEAVE REMAINDER

BE00 46F0 SINF3 LSR SGNFLG ; WAS QUAD ODD
BE02 900D ^BE11 BCC SINF4 ; NO
BE04 286D JSR PMOVE
BE07 A271 LDX #FPONE&$FF
BE09 A9BE LDY #FPONE/$100
BE0B 280DD JSR FLDOF
BE0E 286D JSR PSUB
BE11 SINF4

BE11 A2E6 LDX #FPSCR&$FF ; SAVE ARG
BE13 A005 LDY #FPSCR/$100
```

243
BE15 20A7DD JSR FST0R ; X→FR0
BE16 20B6DD JSR FMOVE ; X**2→FR0
BE17 20BDDB JSR FMUL  
BE1E B085 'BDA5 BCS SINERR  
BE20 A986 LDA #$NSCF  
BE22 A241 LDX #$SCOF&$FF  
BE24 A0BE LDY #$SCOF/$100  
BE26 20BDDB JSR FLYEVL ; EVALUATE P[X**2]  
BE29 A2B6 LDX #$FPSCR&$FF  
BE2B A0A5 LDY #$FPSCR/$100  
BE2D 2098DD JSR FLD1R ; X→FR1  
BE30 20DBDA JSR FMUL  ; SIN[X] = X*P[X**2]  
BE33 46FF LSR SGNFLG  ; WAS QUAD 2 OR 3?  
BE35 0809 'BE40 BCC SINDON ; NO - THRU  
BE37 10 CLC  ; YES  
BE38 A5D4 LDA FR0  ; FLIP SIGN  
BE3A F004 'BE40 BEQ SINDON ; [UNLESS ZERO]  
BE3C 4980 EOR #$80  
BE3E 05D4 STA FR0  
BE40 60 SINDON RTS ; RETURN  
BE41 BDB3551499 SCOFE .BYTE $BD,$03,$55,$14,$89,$39 ; -0000355149939
39  
BE47 3E81604427 .BYTE $3E,$81,$60,$84,$27,$52 ; 0.000160442752  
BE4D BE46817543 .BYTE $BE,$46,$81,$75,$43,$55 ; -004681754355  
55  
BE53 3F07969262 .BYTE $3F,$07,$96,$92,$62,$39 ; 0.0796926399  
39  
BE59 BF64596408 .BYTE $BF,$64,$59,$64,$08,$67 ; -0.6459640867  
67  
BE5F 4001570796 PIOV2 .BYTE $40,$01,$57,$07,$96,$32 ;PI/2  
32  
= 0006 NSCF EQU (+-SCOF)/FPREC  
BE65 4090000000 .BYTE $40,$90,0,0,0,0 ; 90 DEG  
00  
BE6B 3F07145329 PIOV18 .BYTE $3F,$07,$14,$53,$29,$25 ;PI/180  
25  
BE71 4001000000 FPONE .BYTE $40,1,0,0,0,0 ; 1.0  
60  

ATAN[X] — Arctangent

BE77  A900 ATAN LDA #0 ; ARCTAN[X]  
BE79  05FF STA SGNFLG ; SIGN FLAG OFF  
BE7B  85FF STA XFMFLG ; & TRANSFORM FLAG  
BE7D A5D4 LDA FR0  
BE7F 297F AND #$7F  
BE81 C940 CMP #$40  ; CHECK X VS 1.0  
BE83 3015 'BE9A BMI ATAN1  ; X<1.0 - USE SERIES DIRECTLY  
BE85 A5D4 LDA FR0  
BE87 29B0 AND #$80  
BE89 05FF STA SGNFLG  ; REMEMBER SIGN  
BE8B E6F1 INC XFMFLG  
BE8D A97F LDA #$7F  
BE8F 25D4 AND FR0  
BE91 85D4 STA FR0  ; FORCE PLUS  
BE93 A2EA LDX #$FP9S&$FF  
BE95 A0DF LDY #$FP9S/$100  
BE97 2095DE JSR XFORM ; CHANGE ARG TO [X-1]/[X+1]  
BE9A ATAN1  
BE9A A286 LDX #$FPSCR&$FF ; ARCTAN[X], -1<X<1 BY SERIES  
OF APPROXIMATIONS  
BE9C A085 LDY #$FPSCR/$100  ; X→FPSCR  
BE9E 20A7DD JSR FST0R ; X→FR1  
BEA1 20B6DD JSR FMOVE  ; X→FR0  
BEA4 20BDDB JSR FMUL  
BEA7 B039 'BEE2 BCS ATNOUT ; 0° FLOW  
BEAB A90B LDA #$NATCF  
BEAD A2AE LDX #$ACOF&$FF  
BEAD A0DF LDY #$ACOF/$100  

244
Source Code

```
BEAF 2040DD JSR PLYEVL ;P[X*X]
BEB2 B02E "BEE2 BCS ATNOUT
BEF4 A265 LDS #FSCR&$FF
BEF6 A055 LDS #FSCR/$100 
BEF8 2098DD JSR FLD1R ;X->FR1
BEFI 2080DD JSR FMUL ;*P[X*X]
BEF4 A5P1 LDS XPMFLG 
BEF2 F010 "BED4 BEQ ATN2 
; 0 FLOW
BEF5 A2F0 LDS #PIOV4/$FF YES-SUB
BEF8 A005 LDS #PIOV4/100
BEF4 2094DD JSR FLD1R
BEF6 A265 LDS ADD
BEF8 A5P0 LDS SGNFLG
BEF2 85D4 ORA FR0 ; ATAN[-X] = - ATAN[X]
BED4 A5PB STA FR0 ; RADIANS OR DEGREES
BEF6 F00A "BFL4 BEQ ATNOUT ; RADIANS - FINI
BED8 A265 LDS PIOV18/$FF ; DEG - DIVIDE BY PI/180
BEDA A0BE LDS #PIOV18/100
BEDC 2096DD JSR FLD1R
BEFD 2082DD JSR FDIV
BEE2 60 ATNOUT RTS

SQR[X] — Square Root

BEE3 30 SQRERR SEC ;SET FAIL
BEE4 60 RTS

BEE5 A900 SQR STA LDA #0 
BEE7 85F1 STA XPMFLG
BEE9 A5D4 LDS FR0 
BEEB 30FP "BEE3 STA SQRERR
BEED C93F CMP #$3F 
BEEF F017 "BF08 BEQ FSQR ; X IN RANGE OF APPROX - GO DO
BEF1 18 CLC
BEF2 60F1 ADC #1 
BEF4 85F1 STA XPMFLG ; NOT IN RANGE - TRANSFORM
BEF6 85F0 STA FR1 ; MANTISSA = 1
BEF8 A901 LDS M1 
BEFA 85E1 STA FR1+1 
BEFC A2F4 LDS #FREC-2
BEFE A500 LDS #0
BF00 95E2 SQR1 STA FR1+2,X 
BF02 CA DEX
BF03 10FB "BF00 BPL SQR1 
BF05 20BDDB JSR FDIV ; X/100**N 
;SQR[X], 0.1<=X<1.0
BF08 PSQR
BF08 A906 LDS #6 
BF0A 85EF STA SQRCNT
BF0C A265 LDS #FSCR&$FF 
BF0E A005 LDS #FSCR/$100
BF10 2AD7DD JSR PST0R 
BF12 2086DD JSR FSUB 
BF14 A265 LDS #FSCR&$FF ;STASH X IN FSCR 
BF16 A293 LDS #FTWO&$FF ;X->FR1
BF18 A0FB LDS #FTWO/$100 
BF1A 2089DD JSR FLODR ;2.0->FR0
BF1D 2088DD JSR FSUB ;2.0-X 
BF20 A265 LDS #FSCR&$FF ;X->FR1
BF22 A005 LDS #FSCR/$100 
BF24 2089DD JSR FLD1R ;X->FR1 
BF27 2088DD JSR FMUL ;X*2.0-X ;1ST APPROX 
BF2A A265 LDS #FSCR1&$FF
BF2C A505 LDS #FSCR1/$100
BF2E 2087DD JSR PST0R ;Y->FSCR1 
BF31 2086DD JSR FMOVE ;Y->FR1
BF34 A265 LDS #FSCR&$FF 
BF36 A005 LDS #FSCR/$100 
BF38 2089DD JSR FLODR 

245
Source Code

BF3B 2028DB JSR FDIV ;X/Y
BF3E A2EC LDX #$FSCR1&$FF
BF40 A005 LDY #$FSCR1/$100
BF42 2098DD JSR FLD1R ;[X/Y]-Y
BF45 2060DA JSR FSUB
BF48 A26C LDX #$FHALF&$FF
BF4A A005 LDY #$FHALF/$100
BF4C 2099DD JSR FLD1R
BF4F 20B900 JSR FMUL ;0.5*[X/Y]-Y=DELTAY
BF52 A54D LDA FR0 ;DELTAY 0.0
BF54 F06E "BF64 BEQ SQRDON
BF56 A2EC LDX #$FSCR1&$FF
BF5B A005 LDY #$FSCR1/$100
BF5A 2099DD JSR FLD1R
BF5D 2066DA JSR FADD ;Y=Y+DELTAY Y
BF60 C6EF DEC SRQCN ; COUNT & LOOP
BF62 186C "BF2A BPL SQRLP
BF64 A2EC SQRDON LDX #$FSCR1&$FF ; DELTA = 0 - GET Y BACK
BF66 A005 LDY #$FSCR1/$100
BF68 2099DD JSR FLD1R ; WAS ARG TRANSFORMED
BF6B A5F1 , LDA XFMFLG
BF6D F023 "BF92 BEQ SQROUT ; NO FINI
BF6F 30 SEC
BF70 E940 SBC #$40
BF72 18 CLC
BF73 +6A RORA A
BF74 18 CLC
BF75 6940 ADC #$40
BF77 297F AND #$7F
BF79 85E0 STA FRI
BF7B A5F1 LDA XFMFLG
BF7D RORA A
BF7D +6A ROR A
BF7E A901 LDA +1 ; MANTISSA = 1
BF80 9002 "BF84 BCC SQR2 ; WAS EXP ODD OR EVEN
BF82 A910 LDA #$10 ; ODD - MANT = 10
BF84 85E1 SQR2 STA FRI+1
BF86 A204 LDX #$FPREC-2
BF88 A900 LDA +0
BF8A 95E2 SQR3 STA FRI+2,X ; CLEAR REST OF MANTISSA
BF8C CA DEX
BF8D 10FB "BF8A BPL SQR3
BF8F 20B9DA JSR FMUL ; SQR[X] = SQR[X/100**N] * [10**N]
BF92 60 SQROUT RTS
BF93 4002000000 FTWO .BYTE $40,2,0,0,0,0,0 ; 2.0

Floating Point

BF99 = D800 ORG FPORG
D800 LOCAL

ASCIN — Convert ASCII Input to Internal Form

* ON ENTRY INBUFF - POINTS TO BUFFER WITH ASCII
CIX - INDEX TO 1ST BYTE OF #
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
D800 AFP
D800 CVAFP
D800 ASCIN
D800 20A1DB JSR SKPBLANK
D800 20BBDB JSR :TSTCHAR ; SEE IF THIS COULD BE A NUMBER
D803 B039 "D841 BCS :NONUM ; BR IF NOT A NUMBER

246
SET INITIAL VALUES

D808 A2ED LDX #EEXP ; ZERO 4 VALUES
D80A A084 LDY #4 ; X
D80C 2846DA JSR ZXLY ; X
D80F A2FF LDX #$FF
D811 86F1 STX DIGRT ; SET TO #$FF
D813 2044DA JSR ZFR0 ; CLEAR FR0
D816 F004 "D81C BEQ :IN2 ; UNCONDITIONAL BR

D818 :IN1
D81A 85F0 STA FCHRFLG ; X
D81C :IN2
D81F B021 "D842 BCS :NON1 ; BR IF CHAR NOT NUMBER

IT'S A NUMBER

D821 48 PHA SAVE ON CPU STACK
D822 A6D5 LDX FROM GET 1ST BYTE
D824 D011 "D837 BNE :INCE INCR EXPONENT
D826 20EBD8 JSR NIBSH0 SHIFT FR0 ONE NIBBLE LEFT
D829 68 PLA GET DIGIT ON CPU STACK
D82A 05D9 ORA FROM+FMPREC-1 OR INTO LAST BYTE
D82C 85D9 STA FROM+FMPREC-1 SAVE AS LAST BYTE
COUNT CHARACTERS AFTER DECIMAL POINT

D82E A6F1 LDX DIGRT ; GET # OF DIGITS RIGHT
D830 30E6 "D818 BMI :IN1 IF = #$FF, NO DECIMAL POINT
D832 E8 INX ADD IN THIS CHAR
D833 86F1 STX DIGRT ; SAVE
D835 D021 "D818 BNE :IN1 GET NEXT CHAR
INCREMENT # OR DIGIT MORE THAN 9

D837 :INCE
D838 68 PLA CLEAR CPU STACK
D83A A6F1 LDX DIGRT ; HAVE DP?
D83E 1002 "D83E BPL :INCE2 IF YES, DON'T INCR E COUNT
D83C E6ED INC EEXP ; INCR EXPONENT
D83E 4C18BD8 JMP :IN1 GET NEXT CHAR

D841 :NONUM
D841 60 RTS ; RETURN FAIL

NON-NUMERIC IN NUMBER BODY

D842 :NON1
D842 C92E CMP #'.'; IS IT DECIMAL POINT?
D844 F014 "D85A BEQ :DP ; IF YES, PROCESS IT
D846 C945 CMP #'.E'; IS IT E FOR EXPONENT?
D848 F019 "D863 BEQ :EXP ; IF YES, DO EXPONENT
D84A A6F0 LDX FCHRFLG ; IS THIS THE 1ST CHAR
D84C D068 "D886 BNE :EXIT ; IF NOT, END OF NUMERIC INPUT
D84E C92B CMP #'.+'; IS IT PLUS?
Source Code

DB58 F0C6 *D818  BEQ :IN1 ; GO FOR NEXT CHAR
DB52 C92D  CMP #'-' ; IS IT MINUS?
DB54 F000 *D856  BEQ :MINUS

DB56 85EE  STA NSIGN ; SAVE SIGN FOR LATER
DB58 F0BE *D818  BEQ :IN1 ; UNCONDITIONAL BR FOR NEXT CHAR

DB5A A6F1  LDX DIGRT ; IS DIGIT STILL = FF?
DB5C 1058 *D8B6  BPL :EXIT ; IF NOT, ALREADY HAVE DP
DB5E 88  INX DIGRT ; INCR TO ZERO
DB5F 86F1  STX DIGRT ; SAVE

DB61 F0B5 *D818  BEQ :IN1 ; UNCONDITIONAL BR FOR NEXT CHAR

DB63 A5F2  LDA CIX ; GET INDEX
DB65 85EC  STA FX ; SAVE
DB67 2094DB  JSR :GETCHAR ; GET NEXT CHAR
DB6A B037 *D8A3  BCS :NON2 ; BR IF NOT NUMBER

IT'S A NUMBER IN AN EXPONENT

DB6C A5F2  LDA CIX ; GET INDEX
DB6D A5ED  LDA EEXP ; GET # OF CHAR OVER 9
DB6F 4B  PHA ; SAVE
DB72 2094DB  JSR :GETCHAR ; GET NEXT CHAR

DB75 B017 *D8BE  BCS :EXP3 ; IF NOT # NO SECOND DIGIT
DB77 48  PHA ; SAVE SECOND DIGIT

DB78 A5ED  LDA EEXP ; GET 1ST DIGIT
DB7A ASLA ; GET DIGIT * 10
DB77 85ED  STA EEXP ; X
DB7D ASLA ; X
DB77 +0A  ASL A ; X
DB7F 65ED  ADC EEXP ; SAVE
DB81 85ED  STA EEXP ; GET SECOND DIGIT
DB83 68  PLA ; GET EXPONENT INPUTTED
DB84 18  CLC ; SAVE
DB85 65ED  ADC EEXP ; GET EXPONENT INPUTTED
DB87 85ED  STA EEXP ; SAVE

DB89 A4F2  LDY CIX ; INC TO NEXT CHAR
DB8B 209DDB  JSR :CHR1 ; X

DB8E :EXP3

DB8E A5EF  LDA ESIGN ; GET SIGN OF EXPONENT
DB90 F0B5 *D89B  BEQ :EXP1 ; IF NO SIGN, IT IS +
DB92 A5ED  LDA EEXP ; GET EXPONENT ENTERED
DB94 49FF  EOR #$FF ; COMPLEMENT TO MAKE MINUS
DB96 18  CLC ; X
DB97 6901  ADC #1 ; X
DB99 85ED  STA EEXP ; SAVE
DB9B :EXP1

DB9B 68  PLA ; GET # DIGITS MORE THAN 9
DB9C 18  CLC ; CLEAR CARRY
DB9D 65ED  ADC EEXP ; ADD IN ENTERED EXPONENT
DB9F 85ED  STA EEXP ; SAVE EXPONENT
DBA1 D013 *D8B6  BNE :EXIT ; UNCONDITIONAL BR

248
Source Code

; NON-NUMERIC IN EXPONENT
; IS IT PLUS?
DBA3 C92B CMP '#'+' ; IF YES BR
DBA5 F006 "DBAD BEQ :EPLUS ; IS IT A MINUS?
DBA7 C92D CMP '#'-' ; IF NOT, BR
DBA9 D087 "DBB2 BNE :NOTE

; E NOT PART OF OUR #
DBAB :EMIN STA ESIGN ; SAVE EXPONENT SIGN
DBAD :PLUS
DBAD 2094DB JSR :GETCHAR ; GET CHARACTER
DBB0 90BA "DB6C BCC :EXP2 ; IF A #, GO PROCESS EXPONENT

; E NOT PART OF OUR #
DBB2 :NOTE
DBB2 A5EC LDA FRX ; POINT TO 1 PAST E
DBB4 85F2 STA CIX ; RESTORE CIX

; FALL THRU TO EXIT
; WHOLE # HAS BEEN INPUTTED
DBB6 :EXIT

; BACK UP ONE CHAR
DBB6 C6F2 DEC CIX ; DECREMENT INDEX

; CALCULATE POWER OF 10 = EXP - DIGITS RIGHT
; WHERE EXP = ENTERED EXPONENT [COMPLEMENT OF -]
+ # DIGITS MORE THAN 9
DBB8 A5ED LDA EEXP ; GET EXPONENT
DBBA A6F1 LDX DIGRT ; GET # DIGITS RIGHT OF DECIMAL
DBBC 3005 "DBC3 BMI :EXIT1 ; NO DECIMAL POINT
DBBE F003 "DBC3 BEQ :EXIT1 ; # OF DIGITS AFTER D.P.=0
DBC0 38 SEC ; GET EXP - DIGITS RIGHT
DBC1 B5F1 SBC DIGRT ; X

; SHIFT RIGHT ALGEBRAIC TO DIVIDE BY 2 = POWER OF 100
DBC3 :EXIT1
DBC3 48 PHA
DBC4 ROLA ; SET CARRY WITH SIGN OF EXPONENT

; GET EXPONENT AGAIN
DBC4 +2A ROL A ; SHIFT RIGHT
DBC5 68 PLA
DBC6 RORA
DBC6 +6A ROR A
DBC7 85ED STA EEXP ; SAVE POWER OF 100
DBC9 9003 "DBCE BCC :EVEN ; IF NO CARRY # EVEN

; ELSE SHIFT 1 NIBBLE LEFT
DBCB 20EBDB JSR NIBSH0
DBCE :EVEN
DBCE A5ED LDA EEXP ; ADD 40 FOR EXCESS 64 + 4

; FOR NORM
; X

; SAVE AS EXPONENT
DBD0 18 CLC
DBD1 6944 ADC #$44 ; X
DBD3 85D4 STA F00 ; SAVE AS EXPONENT

; NORMALIZE NUMBER
DBD5 2000DC JSR NORM
DBD8 B00B "DBE5 BCS :IND2 ; IF CARRY SET, IT'S AN ERROR
Source Code

; SET MANTISSA SIGN
; DBDA A6EE LDX NSIGN ; IS SIGN OF # MINUS?
DBDC F006 'D8E4 BEQ :INDON ; IF NOT, BR
; DBDE A5D4 LDA FR0 ; GET EXPONENT
DBDE $0900 ORA #$80 ; TURN ON MINUS # BIT
DBDE 85D4 STA FR0 ; SET IN FR0 EXP
DBDE :INDON
DBDE 16 CLC ; CLEAR CARRY
DBDE :IND2
DBDE 60 RTS

FPASC — Convert Floating Point to ASCII

; ON ENTRY FR0 - # TO CONVERT
; * ON EXIT INBUFF - POINTS TO START OF # HIGH ORDER BIT OF LAST BYTE IS ON
;
; DBE6 CVFASC
DBE6 FASC
; DBE6 2051DA JSR INTLBF ;SET INBUFF TO PT TO LBUFF
; DBE9 A930 LDA '0' ; GET ASCII ZERO
DBEB 8D7F05 STA LBUFF-1 ; PUT IN FRONT OF LBUFF
; ; TEST FOR E FORMAT REQUIRED
; DBE9 A5D4 LDA FR0 ; GET EXPONENT
DBEF F026 'D91A BEQ :EXP0 ; IF EXP = 0, #$ = 0, SO BR
DBE2 297F AND #$7F ; AND OUT SIGN
; * * * PROCESS NOT E FORMAT
; DBE6 3B SEC ; SET CARRY
DBEB E93F SBC #$3F ; GET DECIMAL POSITION
DBEF 2070DC JSR :CVFR0 ; CONVERT FR0 TO ASCII CHAR
; D902 2B4ADC JSR :FNZERO ; FIND LAST NON-ZERO CHARACTER
; D905 0980 ORA #$80 ; TURN ON HIGH ORDER BIT
; D907 9D8005 STA LBUFF,X ; STORE IT BACK IN BUFFER
; D90A AD8005 LDA LBUFF ; GET 1ST CHAR IN LBUFF
D90D C92E CMP '.' ; IS IT DECIMAL?
D90F F003 'D914 BEQ :FN6 ; BR IF YES
D911 4C88D9 JMP :FN5 ; ELSE JUMP
; ; EXPONENT IS ZERO - # IS ZERO
; * * *
; D917 4C90D9 JMP :FN4 ; DO FINAL ADJUSTMENT
; * *
; D91A :EXP0
; D91A A9B0 LDA #$80+$30 ; GET ASCII 0 WITH MSB = 1
D91C 8D8005 STA LBUFF ; PUT IN BUFFER
; D91F 60 RTS
; ; PROCESS E FORMAT
; D920 :EFORM
D920 A901 LDA #1 ; GET DECIMAL POSITION
D922 2070DC JSR :CVFR0 ; CONVERT FR0 TO ASCII IN LBUFF

250
D925  28A4DC  JSR  :FNZERO  ; GET RID OF TRAILING ZEROS
D928  E8     INX  CIX  ; INCR INDEX
D929  86F2   STX  ; SAVE INDEX TO LAST CHAR

; ADJUST EXPONENT
D92B  A5D4  LDA  FR0  ; GET EXPONENT
D92D  +8CA  ASLA  A  ; MUL BY 2 [GET RID OF SIGN TOO]
D92E  3B    SEC
D92F  E980  SBC  #$40*2  ; SUB EXCESS 64
D931  AE805  LDX  LBUFF  ; GET 1ST CHAR IN LBUFF
D934  E830  CPX  #0  ; IS IT ASCII 0?
D936  F017 "D94F  BEQ  :EF1  ; PUT DECIMAL AFTER 1ST CHAR [IT'S AFTER 2ND NOW]

D938  AE8105  LDX  LBUFF+1  ; SWITCH D.P. + 2ND DIGIT
D93B  AC8205  LDY  LBUFF+2  ; X
D93E  88E205  STX  LBUFF+2  ; X
D941  88C105  STY  LBUFF+1  ; X

D944  A6F2  LDX  CIX  ; IF CIX POINTS TO D.P.
D946  E002  CPX  #2  ; THEN INC
D948  D002 "D94C  BNE  :NOINC  ; X
D94A  E6F2  INC  CIX  ; X

D94C  :NOINC
D94C  18    CLC  ; X
D94D  6901  ADC  #1  ; X

; CONVERT EXP TO ASCII
D94F  :EF1
D94F  85ED  STA  EEXP  ; SAVE EXPONENT
D951  A945  LDA  #'E'  ; GET ASCII E
D953  A4F2  LDY  CIX  ; GET POINTER
D955  289FDC  JSR  :STCHAR  ; STORE CHARACTER
D958  84F2  STY  CIX  ; SAVE INDEX

D95A  A5ED  LDA  EEXP  ; GET EXPONENT
D95C  100B "D969  BPL  :EPL  ; BR IF PLUS

; EXPONENT IS MINUS - COMPLEMENT IT

D95E  A900  LDA  #0  ; SUBTRACT FROM 0 TO COMPLEMENT
D960  3B    SEC
D961  E5ED  SBC  EEXP  ; X
D963  85ED  STA  EEXP  ; X

D965  A92D  LDA  #'-'  ; GET A MINUS
D967  D002 "D96B  BNE  :EF2  ;
D969  :EPL
D969  A92B  LDA  #'+'  ; GET A PLUS
D96B  :EF2
D96B  289FDC  JSR  :STCHAR  ; STORE A CHARACTER

D96E  A200  LDX  #0  ; SET COUNTER FOR # OF TENS
D970  A5ED  LDA  EEXP  ; GET EXPONENT

D972  :EF3
D972  3B    SEC
D973  E90A  SBC  #10  ; SUBTRACT 10
Source Code

D975 9003 'D97A  BCC :EF4 ; IF < Ø, BRANCH
D977 E8  INX ; INCR # OF 10'S
D978 D0F8 'D972 BNE :EF3 ; BR UNCONDITIONAL
; D97A :EF4
D97A 18  CLC ; ADD BACK IN 10
D97B 696A  ADC #10 ; X
D97D 48  PHA ; SAVE
;
D97E 8A  TXA ; GET # OF 10'S
D97F 209DDC JSR :STNUM ; PUT 10'S IN EXP IN BUFFER
D982 68  PLA ; GET REMAINDER
D983 Ø980  ORA #$80 ; TURN ON HIGH ORDER BIT
D985 209DDC JSR :STNUM ; PUT IN BUFFER
;
FINAL ADJUSTMENT
;
D988 :FN5
D988 AD8005 LDA LBUFF ; GET 1ST BYTE IN LBUFF
[OUTPUT]
D988 C930 CMP #'0' ; IS IT ASCII 0?
D98D D000 'D99C BNE :FN4 IF NOT BR
;
INCREMENTS INBUFF TO POINT TO NON-ZERO
;
D98F 18  CLC ; ADD 1 TO INBUFF
D990 A5F3  LDA INBUFF ; X
D992 6991  ADC #1 ; X
D994 85F3  STA INBUFF ; X
D996 A5F4  LDA INBUFF+1 ; X
D998 6990  ADC #0 ; X
D99A 85F4  STA INBUFF+1 ; X
D99C :FN4
D99C A5D4  LDA FR0 ; GET EXPONENT OF #
D99E 1089 'D9A9 BPL :FADONE ; IF SIGN +, WE ARE DONE
;
D9A0 20C1DC JSR :DECINB ; DEC INBUFF
D9A3 A000  LDY #Ø ; GET INDEX
D9A5 A92D  LDA '#-'; GET ASCII -
D9A7 91F3  STA [INBUFF],Y ; SAVE - IN BUFFER
;
D9A9 :FADONE
D9A9 60 RTS

IFP — Convert Integer to Floating Point

* ON ENTRY  FRØ — CONTAINS INTEGER
* * ON EXIT  FRØ — CONTAINS FLOATING POINT #
* * *
D9AA CVIFP
D9AA IFP
;
MOVE INTEGER AND REVERSE BYTES
;
D9AA A5D4  LDA FRØ ; GET INTEGER LOW
D9AC 85F8  STA ZTEMP4+1 ; SAVE AS INTEGER HIGH
D9AE A5D5  LDA FRØ+1 ; GET INTEGER HIGH
D9B0 85F7  STA ZTEMP4 ; SAVE AS INTEGER LOW
;
D9B2 2844DA JSR ZFRØ ; CLEAR FRØ
D9B5 F8  SED ; SET DECIMAL MODE
*
* DO THE CONVERT
*
*
D9B6 A010  LDY #16 ; GET # BITS IN INTEGER
D9BB :IPP1
D9BB 06F8  ASL ZTEMP4+1 ; SHIFT LEFT INTEGER LOW
D9BA 26F7  ROL ZTEMP4 ; SHIFT LEFT INTEGER HIGH
Source Code

D9BC A283  LDX  #3 ; CARRY NOW SET IF THERE WAS A BIT
D9BE  :IPF2 ; BIGGEST INTEGER IS 3 BYTES

DOUBLE # AND ADD IN 1 IF CARRY SET ;
D9BE B5D4  LDA  FR0,X ; GET BYTE
D9C0  75D4  ADC  FR0,X ; DOUBLE [ADDING IN CARRY FROM SHIFT ;
D9C2  95D4  STA  FR0,X ; SAVE
D9C4  CA   DEX   ; DECREMENT COUNT OF FR0 BYTES
D9C5 D0F7 "D9BE BNE :IPF2 ; IF MORE TO DO, DO IT
D9C7 B8   DEY   ; DECREMENT COUNT OF INTEGER DIGITS
D9C8 D0EE "D9B8 BNE :IPF1 ; IF MORE TO DO, DO IT
D9CA D8   CLD   ; CLEAR DECIMAL MODE

SET EXPONENT ;
D9CB A942  LDA  #$42 ; INDICATE DECIMAL AFTER LAST DIGIT
D9CD 85D4  STA  FR0 ; STORE EXponent
D9CF 4C00DC JMP NORM ; NORMALIZE

FPI — Convert Floating Point to Integer

* ON ENTRY  FR0 — FLOATING POINT NUMBER
* ON EXIT  FR0 — INTEGER
* CC SET  CARRY CLEAR — NO ERROR
* CARRY SET — ERROR
*
D9D2  FPI
*
D9D2 A900  LDA  #0 ; CLEAR INTEGER RESULT
D9D4  85F7  STA  ZTEMP4
D9D6  85F8  STA  ZTEMP4+1

CHECK EXPONENT
*
D9D8 A5D4  LDA  FR0 ; GET EXPONENT
D9DA 3066 "DA42 BMI :ERVAL ; IF SIGN OF FP# IS -, THEN ERROR
D9DC C943  CMP  #$43 ; IS FP# TOO BIG TO BE INTEGER
D9DE B062 "DA42 BCS :ERVAL ; IF YES, THEN ERROR
D9E0  38   SEC   ; SET CARRY
D9E1 E940  SBC  #$40 ; IS FP# LESS THAN 1?
D9E3 903F "DA24 BCC :ROUND ; IF YES, THEN GO TEST FOR ROUND

GET # OF DIGITS TO CONVERT = [EXPONENT -40+1]*2
[A CONTAINS EXPONENT -40]
[CARRY SET]
*
D9E5 6900  ADC  #0 ; ADD IN CARRY
D9E7  A56 ; MULT BY 2
D9E7 +0A  ASL  A
D9E8  85F5  STA  ZTEMP1 ; SAVE AS COUNTER
*
DO CONVERT
*
D9EA  :FPI1
*
MULT INTEGER RESULT BY 10

D9EA 285ADA JSR :ILSHFT ; GO SHIFT ONCE LEFT
D9ED B053 "DA42 BCS :ERVAL ; IF CARRY SET THEN # TOO BIG

D9E5 A5F7 LDA ZTEMP4 ; SAVE INTEGER *2
D9F1 85F9 STA ZTEMP3 ; X
D9F3 A5F8 LDA ZTEMP4+1 ; X
D9F5 85FA STA ZTEMP3+1 ; X

D9F7 285ADA JSR :ILSHFT ; MULT BY *2
D9FA B046 "DA42 BCS :ERVAL ; # TOO BIG
D9FC 285ADA JSR :ILSHFT ; MULT BY *2 [NOW *4 IN ZTEMP4]
D9FF B041 "DA42 BCS :ERVAL ; BR IF # TOO BIG

DA01 18 CLC ; ADD IN *2 TO =*10
DA04 A5F8 LDA ZTEMP4+1 ; X
DA06 85F8 ADC ZTEMP3+1 ; X
DA08 A5F7 LDA ZTEMP4 ; X
DA0A 85F9 ADC ZTEMP3 ; X
DA0C 85F7 STA ZTEMP4 ; X

ADD IN NEXT DIGIT

DA10 28B9DC JSR :GETDIG ; GET DIGIT IN A
DA13 18 CLC
DA14 85F8 ADC ZTEMP4+1 ; ADD IN DIGIT
DA16 85F8 STA ZTEMP4+1 ; X
DA18 A5F7 LDA ZTEMP4 ; X
DA1A 6900 ADC #0 ; X
DA1C B024 "DA42 BCS :ERVAL ; BR IF OVERFLOW
DA1E 85F7 STA ZTEMP4 ; X

DA20 C6F5 DEC ZTEMP1 ; DEC COUNTER OF DIGITS TO DO
DA22 D0C6 "D9EA BNE :FPII ; IF MORE TO DO, DO IT

ROUND IF NEEDED

DA24 :ROUND
DA24 28B9DC JSR :GETDIG ; GET NEXT DIGIT IN A
DA27 C905 CMP #5 ; IS DIGIT <5?
DA29 900D "DA38 BCC :NR ; IF YES, DON'T ROUND
DA2B 18 CLC ; ADD IN 1 TO ROUND
DA2C A5F8 LDA ZTEMP4+1 ; X
DA2E 6901 ADC #1 ; X
DA30 85F8 STA ZTEMP4+1 ; X
DA32 A5F7 LDA ZTEMP4 ; X
DA34 6900 ADC #0 ; X
DA36 85F7 STA ZTEMP4 ; X

MOVE INTEGER TO FR3

DA38 :NR
DA38 A5F8 LDA ZTEMP4+1 ; GET INTEGER LOW
DA3A 85D4 STA FR3 ; SAVE
DA3C A5F7 LDA ZTEMP4 ; GET INTEGER HIGH
DA3E 85D5 STA FR3+1 ; SAVE

DA40 18 CLC ; CLEAR CC FOR GOOD RETURN
DA41 60 RTS

DA42 :ERVAL
DA42 38 SEC ; SET CARRY FOR ERROR RETURN
DA43 60 RTS

* ZFR3 - ZERO FR3
* ZF1 - ZERO 6 BYTES AT LOC X
Source Code

* ZLY - ZERO PAGE ZERO LOC X FOR LENGTH Y *

DA44 ZFRØ
DA44 A2D4 LDX #FRØ ; GET POINTER TO FR1
DA46 ZF1
DA46 A086 LDY #6 ; GET # OF BYTES TO CLEAR
DA48 ZXY
DA48 A90Ø LDA #Ø ; CLEAR A
DA4A :ZF2
DA4A 95ØØ STA 0,X ; CLEAR A BYTE
DA4C E8 INX ; POINT TO NEXT BYTE
DA4D 88 DEY ; DEC COUNTER
DA4E D0FA ~DA4A BNE :ZF2 ; LOOP
DA50 6Ø RTS

; INTLBLF - INIT LBUFF INTO INBUFF ;

DA51 INTLBLF
DA51 A905 LDA #LBUFF/256
DA53 85F4 STA INBUFF+1
DA55 A90Ø LDA #LBUFF&255
DA57 85F3 STA INBUFF
DA59 6Ø RTS

* :ILSHFT - SHIFT INTEGER IN ZTEMP4 LEFT ONCE *

DA5A :ILSHFT
DA5A 18 CLC CLEAR CARRY
DA5B 26F8 ROL ZTEMP4+1 ; SHIFT LOW
DA5D 26F7 ROL ZTEMP4 ; SHIFT HIGH
DA5F 6Ø RTS

Floating Point Routines

FADD — Floating Point Add Routine

ADDS VALUES IN FRØ AND FR1

* ON ENTRY FRØ & FR1 - CONTAIN # TO ADD
* ON EXIT FRØ - RESULT

FSUB — Floating Point Subtract Routine

SUBTRACTS FR1 FROM FRØ

* ON ENTRY FRØ & FR1 - CONTAIN # TO SUBTRACT
* ON EXIT FRØ - RESULT
* BOTH RETURN WITH CC SET:
* CARRY SET IF ERROR
* CARRY CLEAR IF NO ERROR

DA60 FSUB
DA60 A5EØ LDA FR1 ; GET EXPONENT OF FR1
DA62 49ØØ EOR #$8Ø ; CHANGE SIGN OF MANTISSA
DA64 85EØ STA FR1 ; SAVE EXPONENT
; ;
DA66 FADD
DA66 :FRADD
Source Code

```
DA66 A5E0  LDA FR1      ; GET EXPONENT FR1
DA68 297F  AND #$7F    ; TURN OFF MANTISSA SIGN BIT
DA6A 85F7  STA ZTEMP4  ; SAVE TEMPORARILY
DA6C A5D4  LDA FR0     ; GET EXPONENT FR0
DA6E 297F  AND #$7F    ; TURN OFF MANTISSA SIGN BIT
DA70 3B    SEC          ; CLEAR CARRY
DA71 E5F7  SBC ZTEMP4  ; SUB EXPONENTS
DA73 1810 "DA85 BPL :NSWAP ; IF EXP[FR0] >= EXP[FR1], NO SWAP

; SWAP FR0 AND FR1
; DA75 A205 LDX #FMPREC ; GET INDEX
; DA77 :SWAP

DA77 B5D4  LDA FR0,X    ; GET BYTE FROM FR0
DA79 B4E0  LDY FR1,X    ; GET BYTE FROM FR1
DA7B 95E0  STA FR1,X    ; PUT FR0 BYTE IN FR1
DA7D 98    TYA          ; GET FR1 BYTE
DA7F 95D4  STA FR0,X    ; PUT FR1 BYTE IN FR0
DA80 CA    DEX          ; DEC INDEX
DA81 18F4 "DA77 BPL :SWAP ; IF MORE TO DO, GO SWAP
DA83 38E1 "DA66 BMI :FRADD ; UNCONDITIONAL

DA85 F007 "DA8E BEQ :NALIGN ; IF DIFFERENCE = 0, ALREADY
DA87 C905  CMP #FMPREC ; IS DIFFERENCE < # OF BYTES
DA89 B019 "DA44 BCS :ADDEND ; IF NOT, HAVE RESULT IN FR0

; DAA8 203EDC
; JXR RSHT1 ; SHIFT TO ALIGN

; DAB0 4C00DC JMP NORM
; ADD IN FIND CARRY

; DAB3 A204 LDX #FMPREC-1 ; GET POINTER FOR LAST BYTE
; DA97 18 CLC ; CLEAR CARRY
; DA99 :ADD1

; ADD FR0 & FR1

; DA9B B5D5  LDA FR0M,X  ; GET BYTE OF FR0
; DA9D 75E1  ADC FR1M,X  ; ADD IN BYTE OF FR1
; DA9F 95D5  STA FR0M,X  ; STORE
; DA9E CA    DEX          ; DEC POINTER
; DA9F 18F7 "DA98 BPL :ADD1 ; ADD NEXT BYTE

; DAB0 D8    CLD          ; CLEAR DECIMAL MODE
; DAB2 B03B "DAA7 BCS :ADD2 ; IF THERE IS A CARRY, DO IT
; DAB4 :ADDEND

; DAB4 4C00DC JMP NORM ; GO NORMALIZE

; DAB7 A901  LDA #1      ; GET 1 TIMES TO SHIFT
; DAB9 203ADC JSR RSHT0  ; GO SHIFT

; DAB9 A901  LDA #01     ; GET CARRY
; DAAE 85D5  STA FR0M    ; ADD IN CARRY
; DAB0 4C00DC JMP NORM

; SUBTRACT FR1 FROM FR0

; DAB3 :SUB

; DAB3 A204 LDX #FMPREC-1 ; GET POINTER TO LAST BYTE
; DAB5 3B SEC ; SET CARRY
```
Source Code

```
DAB6 B5D5 LDA #FR0, X ; GET FR0 BYTE
DAB8 95D5 STA #FR0, X ; SUB FR1 BYTE
DABC CA DEX ; DEC POINTER
DABD 10F7 DAB6 BPL :SUB1 ; SUB NEXT BYTE
DABF 9004 DAC5 BCC :SUB2 ; IF THERE IS A BORROW DO IT
DAC1 D8 CLD ; CLEAR DECIMAL MODE
DAC2 4C00DC JMP NORM

; TAKE COMPLEMENT SIGN

; COMPLEMENT MANTISSA

DACB 38 SEC ; SET CARRY
DACCC A264 LDX #$F0REC-1 ; GET INDEX COUNTER
DACE :SUB3
DACE A908 LDA #0 ; GET ZERO
DADD 95D5 STA #FR0, X ; COMPLEMENT BYTE
DADA CA DEX ; MORE TO DO
DADD 10F7 DACE BPL :SUB3 ; BR IF YES
DAD7 D8 CLD ; CLEAR DECIMAL MODE
DAD8 4C00DC JMP NORM ; GO NORMALIZE

FMUL — Multiply FR0 by FR1

* ON ENTRY # ARE IN FR0 AND FR1
* * * ON EXIT FR0 — CONTAINS PRODUCT
* RETURN WITH CC SET
* CARRY SET IF ERROR
* CARRY CLEAR IF NO ERROR
* * * * *
DADB FMUL

; SET UP EXPONENT

; DADB A5D4 LDA FR0 ; GET EXP FR0
DADB F045 "DB24 BEQ MEND3 ; IF = 0, DONE
DADF A5E0 LDA FR1 ; GET FR1 EXP
DAD8 F03E "DB21 BEQ MEND2 ; IF =0, ANSWER =0

DAD9 20CFDC JSR MDESUP ; DO COMMON SET FOR EXPONENT
DAD6 38 SEC ; SET CARRY
DAD7 8940 SEC #$40 ; SUB EXCESS 64
DAD9 38 SEC ; SET CARRY TO ADD 1
DADA 65E0 ADD FR1 ; ADD 1 + FR1 EXP TO FR0 EXP
DAEC 3038 "DB26 BMI $EROV ; IF - THEN OVERFLOW

; FINISH MULTIPLY SET UP

; DAEF 20E0DC JSR MDSUP ; DO SET UP COMMON TO DIVIDE

* * *
DAF1 :FRM

* GET # OF TIMES TO ADD IN MULTIPLICAND
```
DAF1  A5DF  LDA  FRE+FMPREC  ; GET LAST BYTE OF FRE
DAF3  290F  AND  #$0F  ; AND OUT HIGH ORDER NIBBLE
DAF5  85F6  STA  ZTEMP+1  ; SET COUNTER FOR LOOP CONTROL
 ;
 ; ADD IN FR1
 ;
DAF7  C6F6  DEC  ZTEMP+1  ; DEC MULT COUNTER
DAF9  30F6  'DB01  BMI  :FRM2  ; IF - THIS LOOP DONE
DAFB  261DD  JSR  FRA10  ; ADD FR1 TO FR0 [6 BYTES]
DAF7  4CF7DA  JMP  :FRM1  ; REPEAT
 ;
 ; GET # OF TIMES TO ADD IN MULTIPLICAND * 10
 ;
DB01  A5DF  LDA  FRE+FMPREC  ; GET LAST BYTE OF FRE
DB03  +4A  LSR  A  ; SHIFT OUT LOW ORDER NIBBLE
DB04  LSRA  ; X
DB05  +4A  LSR  A  ; X
DB06  +4A  LSR  A  ; X
DB07  85F6  STA  ZTEMP+1  ; SAVE AS COUNTER
 ;
 ; ADD IN FR2
 ;
DB09  C6F6  DEC  ZTEMP+1  ; DECREMENT COUNTER
DB0B  30F6  'DB13  BMI  :NXTB  ; IF --, DO NEXT BYTE
DB0D  265DD  JSR  FRA20  ; ADD FR2 TO FR0 [6 BYTES]
DB10  4C06DB  JMP  :FRM3  ; REPEAT
 ;
 ; SET UP FOR NEXT SET OF ADDS
 ;
DB13  :NXTB
 ;
 ; SHIFT FR0/FRE RIGHT ONE BYTE
 [THEY ARE CONTIGUOUS]
 ;
DB13  2062DC  JSR  RHSF0E  ; SHIFT FR0/FRE RIGHT
 ;
 ; TEST FOR # OF BYTES SHIFTED
DB16  C6F5  DEC  ZTEMP1  ; DECREMENT LOOP CONTROL
DB18  DB07  'DAF1  BNE  :FRM  ; IF MORE ADDS TO DO, DO IT
 ;
 ; SET EXPONENT
 ;
DB1A  MDEND
DB1A  A5ED  LDA  EEXP  ; GET EXPONENT
DB1C  85D4  STA  FR0  ; STORE AS FR0 EXP
 ;
DB1E  MEND1
DB1E  4C04DC  JMP  NORM1  ; NORMALIZE
 ;
DB21  MEND2
DB21  2044DA  JSR  ZFR0  ; CLEAR FR0
DB24  MEND3
DB24  18  CLC  ; CLEAR CARRY FOR GOOD RTN
DB25  60  RTS  ;
 ;
DB26  :EROV
DB26  38  SEC  ; SET CARRY FOR ERROR ROUTINE
DB27  60  RTS  ; RETURN
Source Code

FDIV — Floating Point Divide

ON ENTRY  FR0 — DIVIDEND
FR1 — DIVISOR

ON EXIT  FR0 — QUOTIENT

RETURNS WITH CC SET:
CARRY CLEAR — ERROR
CARRY SET — NO ERROR

DB28  FDIV

DO DIVIDE SET UP

DB28  A5E0  LDA  FR1  ; GET FR1 EXP
DB2A  F0FA  DB26  BEQ  : ER0V  ; IF =0, THEN OVERFLOW
DB2C  A5D4  LDA  FR0  ; GET EXPONENT FR0
DB2E  F0F4  DB24  BEQ  MEND3  ; IF = 0, THEN DONE

DB30  20CFDC  JSR  MDESUP  ; DO COMMON PART OF EXP SET UP

DB33  38  SEC
DB34  E5E0  SEC  FR1  ; SUB FR1 EXP FROM FR0 EX
DB36  18  CLC
DB37  6940  ADC  #$40  ; ADD IN EXCESS 64
DB39  30E2  DB26  BMI  : ER0V  ; IF MINUS THEN OVERFLOW

DB3B  20E0DC  JSR  MDSUP  ; DO SETUP COMMON FOR MUL
DB3E  E6F5  INC  ZTEMP1  ; LOOP 1 MORE TIME FOR DIVIDE
DB40  4C4EDB  JMP  : FRD1  ; SKIP SHIFT 1ST TIME THROUGH

= 00D9  GTEMP  EQU  FR0+FMPREC

DB43  : NXTQ

SHIFT FR0/FRE LEFT ONE BYTE
[THEY ARE CONTIGUOUS]

DB43  A200  LDX  #$0  ; GET POINTER TO BYTE TO MOVE
DB45  : NXTQ1

DB45  B5D5  LDA  FR0+1,X  ; GET BYTE
DB47  95D4  STA  FR0,X  ; MOVE IT LEFT ONE BYTE

DB49  E9  INX  ; POINT TO NEXT BYTE
DB4A  E08C  CPX  #$FMPREC*2+2  ; HAVE WE DONE THEM ALL?
DB4C  D0F7  DB45  BNE  : NXTQ1  ; IF NOT, BRANCH

DO DIVIDE

DB4E  : FRD1

SUBTRACT FR2 [DIVISOR * 2] FROM FRE [DIVIDEND]

DB4E  A005  LDY  #$FMPREC  ; SET LOOP CONTROL
DB50  3B  SEC  ; SET CARRY
DB51  F8  SED  ; SET DECIMAL MODE
DB52  : FRS2

DB52  B9DA00  LDA  FRE,Y  ; GET A BYTE FROM FRE
DB55  F9E600  SBC  FR2,Y  ; SUB FR2
DB58  99DA00  STA  FRE,Y  ; STORE RESULT
DB5B  8B  DEY  ; DEC COUNTER
DB5C  10F4  DB52  BPL  : FRS2  ; BR IF MORE TO DO
DB5E  D8  CLD  ; CLEAR DECIMAL MODE

DB5F  9004  DB65  BCC  : FAIL  ; IF RESULT <0 [FRE < FR2] BR

DB61  E6D9  INC  GTEMP  ; INCR # TIMES SUB [QUOTIENT]

259
Source Code

```
DB63 DB E9 DB 4E BNE :FRD1 ; SUB AGAIN
; ; SUBTRACT OF FR2 DIDN'T GO
; ;
DB65 :FAIL
DB65 200F DD JSR FRA2E ; ADD FR2 BACK TO FR0
; ; SHIFT LAST BYTE OF QUOTIENT ONE NIBBLE LEFT
; ;
DB68 0E D9 ASL QTEMP ; SHIFT 4 BITS LEFT
DB6A 0E D9 ASL QTEMP ; X
DB6C 0E D9 ASL QTEMP ; X
DB6E 0E D9 ASL QTEMP ; X
DB70 :FRD2
; ;
DB72 A005 LDY #FMPREC ; SET LOOP CONTROL
DB73 F8 SED ; SET DECIMAL MODE
DB74 :FRS1
DB76 B9DA00 LDA FRE,Y ; GET A BYTE FROM FRE
DB77 F9E800 SBC FR1,Y ; SUB FR1
DB79 99D800 STA FRE,Y ; STORE RESULT
DB7B 88 DEY
DB7D 18F4 DB74 SPL :FRS1 ; BR IF MORE TO DO
DB79 9004 DB87 BCC :FAIL2 ; IF RESULT < 0 [FRE < FR1] BR
DB83 E6D9 INC QTEMP ; INCR # TIMES SUB [QUOTIENT]
DB85 D0E9 DB70 BNE :FRD2 ; SUB AGAIN
; ; SUBTRACT OF FR1 DIDN'T GO
; ;
DB87 :FAIL2
DB87 2009 DD JSR FRA1E ; ADD FR1 BACK TO FR0
DB8A C6F5 DEC ZTEMP1 ; DEC LOOP CONTROL
DB8C D0B5 DB43 BNE :NXTQ ; GET NEXT QUOTIENT BYTE
; ;
DB8E 2862 DC JSR RSHF0E ;SHIFT RIGHT FR0/FRE TO CLEAR EXP
DB91 4C1ADB JMP MDEND ; JOIN MULT END UP CODE

:GETCHAR — Test Input Character

* ON ENTRY INBUFF — POINTS TO BUFFER WITH INPUT
CIX — POINTS TO CHAR IN BUFFER
* *
* ON EXIT CIX — POINTS TO NEXT CHAR
CC — CARRY CLEAR IF CHAR IS NUMBER
CARRY SET IF CHAR NOT NUMBER
* *

DB94 :GETCHAR
DB94 20 AFDB JSR TSTNUM ; GO TEST FOR NUMBER
DB97 A4F2 LDY CIX ; GET CHARACTER INDEX
DB99 9002 DB9D BCC :GCHR1 ; IF CHAR = NUM, SKIP
; ;
DB9B 1F3 LDA [INBUFF],Y ; GET CHARACTER
; ;
DB9D :GCHR1
DB9D CB INY ; POINT TO NEXT CHAR
DB9E 84F2 STY CIX ; SAVE INDEX
DBA0 60 RTS
; ; SKPBLANK-SKIP BLANKS
; ; STARTS AT CIX AND SCANS FOR NON BLANKS
```

260
Source Code

```
DBA1  SKBLANK
DBA1  SKPBLANK
DBA1  A4F2  LDY  CIX  ;GET CIX
DBA3  A920  LDA  #$20  ;GET A BLANK

DBA5  D1F3  ;SB1  CMP  [INBUFF],Y  ;IS CHAR A BLANK
DBA7  D033  DJBAC  BNE  ;SBRTS  ;BR IF NOT
DBA9  C8    INY    ;INC TO NEXT
DBAA  D0F9  DJB5   BNE  ;SB1  ;GO TEST

DBA3  S4F2  ;SBRTS  STY  CIX  ;SET NON BLANK INDEX
DBAE  60    RTS  ;RETURN

; TSTNUM-TEST CHAR AT CIX FOR NUM
; - RTNS CARRY SET IF NUM

DBAF  TSTNUM
DBAF  A4F2  LDY  CIX  ;GET INDEX
DBB1  B1F3  LDA  [INBUFF],Y  ;AND GET CHAR
DBB3  38    SEC
DBB4  E930  SBC  #$30  ;SUBTRACT ASCLT ZERO
DBB6  9018  DBD0  BCC  ;TSNFAL  ;BR CHAR<ASCLT ZERO
DBB8  C90A  CMP  #$0A  ; TEST GT ASCLT 9
DBBA  60    RTS  ;DONE

:TSTCHAR — Test to See if This Can Be a Number

*  ON EXIT  CC - CARRY SET IF NOT A #
*  CARRY CLEAR IF A #

DBBB  :TSTCHAR
DBBB  A5F2  LDA  CIX  ;GET INDEX
DBBD  48    PHA  ;SAVE IT
DBBE  B94DB  JSR  :GETCHAR  ;GET CHAR
DBC1  901F  DJB2   BCC  ;RTPASS  ; IF = #8 RETURN PASS

DBC3  C92E  CMP  #'.',  ; IF = D.P., OK SO FAR
DBC5  F014  DJBDB  BEQ  ;TSTN  ; IF = +8 OK SO FAR
DBC7  C92B  CMP  #'+',  ; IF = -8 OK SO FAR
DBC9  F007  DJBD2  BEQ  ;TSTN1
DBCB  C92D  CMP  #'-',  ; IF = -8 OK SO FAR

DBCD  F003  DJBD2  BEQ  ;TSTN1

DBCF  :RTFAIL
DBCF  68    PLA  ; CLEAR STACK
DBD0  38    :TSNFAL  SEC  ;SET FAIL
DBD1  60    RTS

DBD2  :TSTN1
DBD2  B94DB  JSR  :GETCHAR  ;GET CHAR
DBD5  900B  DJB2   BCC  ;RTPASS  ; IF #, RETURN PASS
DBD7  C92E  CMP  #'.',  ; IS IT D.P.
DBD9  D0F4  DJBCF  BNE  ;RTFAIL  ; IF NOT, RETURN FAIL
DBDB  :TSTN
DBDB  294DB  JSR  :GETCHAR  ;ELSE GET NEXT CHAR
DBDE  9002  DJB2   BCC  ;RTPASS  ; IF #, RETURN PASS
DBDO  80ED  DJBCF  BCS  ;RTFAIL  ;ELSE, RETURN FAIL

DBE2  :RTPASS
DBE2  68    PLA  ; RESTORE CIX
DBE3  85F2  STA  CIX  ; X
DBE5  18    CLC  ; CLEAR CARRY
DBE6  60    RTS  ; RETURN PASS

NIBSHO — Shift FR0 One Nibble Left

*  NIBSH2 - SHIFT FR2 ONE NIBBLE LEFT

DBE7  NIBSH2
DBE7  A2E7  LDX  #FR2+1  ; POINT TO 1ST MANTISSA BYTE
```

261
Source Code

```
DBE9 D002 "DBED" BNE :NIB1
; DBEB NIBSH0
DBEB A2D5 LDX #FR0M ; POINT TO MANTISSA OF FR0
DBED A004 LDY #4 ; GET # OF BITS TO SHIFT
DBEF :NIBS
DBEF 18 CLC ; CLEAR CARRY
DBF0 3604 ROL 4,X ; ROLL
DBF2 3609 ROL 2,X ; X
DBF6 3601 ROL 1,X ; X
DBF8 3600 ROL 0,X ; X
DBFA 26EC ROL FRX ; SAVE SHIFTED NIBBLE
; DBFC 88 DEY ; DEC COUNT
DBFD D8F0 "DBEF" BNE :NIBS ; IF NOT = 0, REPEAT
DBFF 60 RTS

NORM — Normalize Floating Point Number

DC00 NORM
DC00 A200 LDX $0 ; GET ZERO
DC02 86DA STX FR0+FPREC ; FOR ADD NORM SHIFT IN A ZERO
DC04 NORM1
DC04 A204 LDX #FMPREC-1 ; GET MAX # OF BYTES TO SHIFT
DC06 A5D4 LDA FR0 ; GET EXPONENT
DC08 F02E "DC38" BEQ :NDONE ; IF EXP=0, # = 0
DC0A :NORM
DC0A A5D5 LDA FR0M ; GET 1ST BYTE OF MANTISSA
DC0C D01A "DC28" BNE :TSTBIG ; IF NOT = 0 THEN NO SHIFT
; SHIFT 1 BYTE LEFT
; DC0E A000 LDY $0 ; GET INDEX FOR 1ST MOVE BYTE
DC10 :NSH
DC10 B9D600 LDA FR0M+1,Y ; GET MOVE BYTE
DC13 99D500 STA FR0M,Y ; STORE IT
DC16 C8 INY ; INC Y-INDEX
DC17 C005 CPY #FMPREC ; ARE WE DONE
DC19 90F5 "DC10" BCC :NSH IF NOT SHIFT AGAIN
; DECREMENT EXPONENT
DC1B C6D4 DEC FR0 ; DECREMENT EXPONENT
DC1D CA DEX ; DEC COUNTER
DC1E D0EA "DC0A" BNE :NORM ; DO AGAIN IF NEEDED
; DC20 A5D5 LDA FR0M ; IS MANTISSA STILL 0
DC22 D004 "DC28" BNE :TSTBIG ; IF NOT, SEE IF TOO BIG
DC24 85D4 STA FR0 ; ELSE ZERO EXP
DC26 18 CLC
DC27 60 RTS
; DC28 :TSTBIG
DC28 A5D4 LDA FR0 ; GET EXPONENT
DC2A 297F AND #$7F ; AND OUT SIGN BIT
DC2C 971 CMP #49+64 ; IS IT < 49+64?
DC2E 9001 "DC31" BCC :TSTUND ; IF YES, TEST UNDERFLOW
DC30 60 RTS ; SO RETURN
DC31 :TSTUND
DC31 C9F0 CMP #-49+64 ; IS IT >=-49+64?
DC33 B003 "DC38" BCS :NDONE ; IF YES, WE ARE DONE
DC35 2044DA JSR ZFR0 ; ELSE # IS ZERO
; DC38 :NDONE
DC38 18 CLC ; CLEAR CARRY FOR GOOD RETURN
DC39 60 RTS
```

262
RSHFT0 — Shift FR0 Right/Increment Exponent

RSHFT1 — Shift FR1 Right/Increment Exponent

* * * ON ENTRY A - # OF PLACES TO SHIFT * * *

DC3A **** RSHFT0
DC3A A2D4 LDX #FR0 ; POINT TO FR0
DC3C D002 ~DC40 BNE :RSH ;

DC3E **** RSHFT1
DC3E A2E0 LDX #FR1 ; POINT TO FR1

DC40 :RSH
DC40 86F9 STX ZTEMP3 ; SAVE FR POINTER
DC42 85F7 STA ZTEMP4 ; SAVE # OF BYTES TO SHIFT
DC44 85F8 STA ZTEMP4+1 ; SAVE FOR LATER

DC46 :RSH2
DC46 A004 LDY #FMPREC-1 ; GET # OF BYTES TO MOVE

DC48 :RSH1
DC48 B504 LDA 4,X ; GET CHAR
DC4A 9505 STA 5,X ; STORE CHAR
DC4C CA DEX ; POINT TO NEXT BYTE
DC4D 88 DEY ; DEC LOOP CONTROL
DC4E DBF8 ~DC48 BNE :RSH1 ; IF MORE TO MOVE, DO IT
DC50 A900 LDA #0 ; GET 1ST BYTE
DC52 9505 STA 5,X ; STORE IT

DC54 A6F9 LDX ZTEMP3 ; GET FR POINTER
DC56 C6F7 DEC ZTEMP4 ; DO WE NEED TO SHIFT AGAIN?
DC58 DBE0 ~DC46 BNE :RSH2 ; IF YES, DO IT

; FIX EXPONENT

DC5A B500 LDA 0,X ; GET EXPONENT
DC5C 18 CLC
DC5D 65F8 ADC ZTEMP4+1 ; SUB # OF SHIFTS
DC5F 9500 STA 0,X ; SAVE NEW EXPONENT
DC61 60 RTS

RSHF0E — Shift FR0/FRE 1 Byte Right [They Are Contiguous]

DC62 RSHF0E
DC62 A20A LDX #FMPREC*2 ; GET LOOP CONTROL ;

DC64 :NXTB1
DC64 B5D4 LDA FR0,X ; GET A BYTE
DC66 95D5 STA FR0+1,X ; MOVE IT OVER 1

DC68 CA DEX ; DEC COUNTER
DC69 18F9 ~DC64 BPL :NXTB1 ; MOVE NEXT BYTE
DC6A A900 LDA #0 ; GET ZERO
DC6C 85D4 STA FR0 ; SHIFT IT IN
DC6E 60 RTS

:CVFR0 — Convert Each Byte in FR0 to 2 Characters in LBUFF

* * * ON ENTRY A - DECIMAL POINT POSITION * * *

DC70 **** CVFR0
DC70 85F7 STA ZTEMP4 ; SAVE DECIMAL POSITION

DC72 A200 LDX #0 ; SET INDEX INTO FR0M
DC74 A000 LDY #0 ; SET INDEX INTO OUTPUT LINE [LBUFF].

; ; CONVERT A BYTE

263
Source Code

DC76 2Ø93DC :CVBYTE
DC76 JSR :TSTDP
DC79 ;PUT IN D.P. NOW?
DC79 :CVB1
DC79 3B SEC
DC7A #1 STA
DC7C ZTEMP4
; DECREMENT DECIMAL POSITION
; X
; SAVE IT

; DO 1ST DIGIT
; DO 1ST DIGIT
; GET FROM FR0
DCB0 +4A LSRA A
DCB1 LSRA A
DCB2 +4A LSRA A
DCB3 LSRA A
DCB4 +4A

DCB5 LDA FROM, X
DCB6 GET FROM FR0
DCB7 B5D5 LDA FROM, X
DCB8 ;GET NUMBER FROM FR0
DCB9 290F AND #$3F
DCB8 AND OUT HIGH ORDER BITS
DCB9 2Ø9DDC JSR :STNUM
DCB10 ;GO PUT # IN BUFFER
DCB9 90E3 'DC76 BCC :CVBYTE IF NOT , MORE TO DO
DCB9 :TSTDP

DCB7 B5D5 LDA FROM, X
DCB8 GET NUMBER FROM FR0
DCB9 2Ø9DDC JSR :STNUM
DCB10 ;GO PUT # IN BUFFER
DCB11 0930 ORA #$30

DC91 90E3 'DC76 BCC :CVBYTE
DC92 # ' . GET ASCII DECIMAL POINT
DC93 E92 ELD A # ' . GET ASCII DECIMAL POINT

DC93 :TSTDP
DC93 2Ø9DDC JSR :STNUM
DC94 ;GO PUT # IN BUFFER
DC95 9980Ø5 STA LBUFF,Y
DC96 BNE ;PUT IN LBUFF
DC97 A92E LDA #'.'
DC98 ;GET ASCII DECIMAL POINT
DC99 2Ø9DDC JSR :STCHAR
DC9A ;PUT D.P. IN BUFFER

:STNUM — Put ASCII Number in LBUFF

* ON ENTRY A - DIGIT TO BE CONVERTED TO ASCII
* AND PUT IN LBUFF
* Y - INDEX IN LBUFF

:STCHAR — Store Character in A in LBUFF

DC9D 093Ø ORA #$3Ø
DC9E :STCHAR
DC9F 998Ø05 STA LBUFF,Y
DC9G BNE ;PUT IN LBUFF

DCA2 CB INY
DCA3 6Ø RTS

:FNZERO — Find Last Non-zero Character in LBUFF

* ON EXIT A - LAST CHAR
* X - POINT TO LAST CHAR
* 

DCA4 :FNZERO
DCA4 AØA LDX #1Ø
DCA5 :PN3
DCA6 BD8ØØ5 LDA LBUFF,X
DCA7 C92E CMP #'.'
DCA8 FØ07 'DCB4 BEQ :PN1
DCA9 C93Ø CMP #'0'
DCA9 C93Ø CMP #0'
DCA10 DØ7 'DCB8 BNE :FN2
DCA11 CA DEX
DCA12 DØ2 ØDCA6 BNE :FN3

264
:GETDIG — Get Next Digit from FR0

* ON ENTRY FR0 - #
* ON EXIT A - DIGIT

:DECINB — Decrement INBUFF

MDESUP — Common Set-up for Multiply and Divide Exponent

MDSUP — Common Set-up for Multiply and Divide
Source Code

DCP2  290F   AND  #50F   ; AND OUT HIGH ORDER NIBBLE
DCP4  8566   STA  FR2   ; STORE TO FINISH SHIFT

DCP6  A905   LDA  #FMPREC   ; SET LOOP CONTROL
DCP8  85F5   STA  ZTEMP1   ; X

DCF4  2844DD  JSR  MVFROE   ; MOVE FR0 TO FRE
DCF6  2044DA  JSR  2FR0   ; CLEAR FR0

DD00  60   RTS

FRA

* FRA10 - ADD FR1 TO FR0 [6 BYTES]
* FRA20 - ADD FR2 TO FR0 [6 BYTES]
* FRA1E - ADD FR1 TO FRE
* FRA2E - ADD FR2 TO FRE

DD01  A2D9   LDX  #FR0+FMPREC   ; POINT TO LAST BYTE OF SUM
DD03  D006  'DD0B  BNE  :F1

DD05  FRA2D
DD05  A2D9   LDX  #FR0+FMPREC
DD07  D00B  'DD11  BNE  :F2

DD09  FRA1E
DD09  A2DF   LDX  #FRE+FMPREC
DD0B  A0E5   LDY  #FR1+FMPREC
DD0D  D004  'DD13  BNE  :FRA

DD0F  FRA2E
DD0F  A2DF   LDX  #FRE+FMPREC
DD11  A0EB   LDY  #FR2+FMPREC

DD13  FRA
DD13  A905   LDA  #FMPREC   ; GET VALUE FOR LOOP CONTROL
DD15  85F7   STA  ZTEMP4   ; SET LOOP CONTROL
DD17  1B   CLC
DD18  F8   SED
DD19  :FRA1

DD19  B500   LDA  0,X   ; GET 1ST BYTE OF
DD1B  790000  ADC  0,Y   ; STORE
DD1E  9500   STA  0,X   ; POINT TO NEXT BYTE
DD20  CA   DEX   ; POINT TO NEXT BYTE
DD21  8B   DEY   ; DEC COUNTER
DD22  C6F7   DEC  ZTEMP4
DD24  10F3  'DD19  BPL  :FRA1   ; IF MORE TO DO, DO IT
DD26  DB   CLD   ; CLEAR DECIMAL MODE
DD27  60   RTS

MVFR12 — Move FR1 to FR2

DD28  MVFR12
DD28  A005   LDY  #FMPREC   ; SET COUNTER
DD2A  :MV2
DD2A  980000  LDA  FR1,Y   ; GET A BYTE
DD2D  99E600  STA  FR2,Y   ; STORE IT

DD30  88   DEY   ; DEC COUNTER
DD31  10F7  'DD2A  BPL  :MV2   ; IF MORE TO MOVE, DO IT
DD33  60   RTS
MVFR0E — Move FRO to FRE

**Polynomial Evaluation**

\[
\]

\[
= \left[\sum_{i=0}^{N} A[i]X^i \right] \text{for } X \in \text{FR0, } N+1 \in \text{A-REG}
\]

* № ENTER 
* № STOT & $FF
* № STOT & $100
* № FSTOR, FMOVE, FLDR, FADD, FMUL
* № STX FPTR2, SAVE POINTER TO COEFF'S
* № FSTR, STM2, STM1, STM0
* № STX FPTR2, SAVE ARG, ARG->FR1

**Floating Load/Store**

DD34 MVFR0E
DD34 A05 LDY #MCPREC
DD36 :MV1
DD36 B9D400 LDA FR0, Y
DD39 99DA00 STA FRE, Y

DD3C 88 DEY
DD3D 18F7 "DD36 BPL :MV1
DD3F 60 RTS

DD40 86FE PLYEVL STX FPTR2 ;SAVE POINTER TO COEFF'S
DD42 84FF STY FPTR2+1
DD44 85EF STA PLYCNT
DD46 A2E0 LDX #PLYARG&$100
DD48 A005 LDY #PLYARG/100
DD4A 20A77D JSR FSPOR ;SAVE ARG
DD4D 20B67D JSR FMODR
DD50 A6FE LDX FPTR2
DD52 A4FF LDY FPTR2+1
DD54 2009DD JSR FLDMR ;COEF->FR0 [INIT SUM]
DD57 C6EF DEC PLYCNT
DD59 F02D 'DD88 BEQ PLYOUT ;DONE ?
DD5B 2986BD PLYEV1 JSR FMUL ; SUM * ARG
DD5E B02B 'DD88 BCS PLYOUT ; 0'FLOW
DD60 1B CLC
DD61 A5FE LDA FPTR2 ;BUMP COEF POINTER
DD63 6906 ADC #FPREC
DD65 85FE STA FPTR2
DD67 9006 'DD6F BCC PLYEV2
DD69 A5FF LDA FPTR2+1 ;ACROSS PAGE
DD6B 9000 ADC #0
DD6D 85FF STA FPTR2+1
DD6F A6FE PLYEV2 LDX FPTR2
DD71 A5FE LDY FPTR2+1 ;GET NEXT COEF
DD73 2986DD JSR FLD1R
DD76 2966DA JSR FADD ;SUM*ARG + COEF
DD79 B00D 'DD88 BCS PLYOUT ; 0'FLOW
DD7B C6EF DEC PLYCNT
DD7D F009 'DD88 BEQ PLYOUT
DD7F A2E0 LDY #PLYARG&$100
DD81 A005 LDY #PLYARG/100
DD83 2986DD JSR FLD1R ;GET ARG AGAIN
DD86 3BDD 'DD5B BMI PLYEV1 ; [=JMP]
DD88 60 PLYOUT RTS

* LOAD FR0 FROM [X,Y] X=LSB, Y=MSB, USES FLPTER [PG0]
DD89 86FC FLDWR STX FLPTR
DD8B 84FD STY FLPTR+1
DD8D A005 FLDWP LXY #FPREC-1 ; #BYTES ENTER HERE W/FLPTR SET
DD8F B1FC FLD01 LDA [FLPTR], Y ; MOVE
DD91 99D400 STA FR0, Y
DD94 88 DEY
DD95 18FB 'DD8P BPL FLD01 ; COUNT & LOOP
DD97 60 RTS

* LOAD FR1 FROM [X,Y] OR [FLPTR]
DD98 86FC FLD1R STX FLPTR ; FLPTR=>[X,Y]
Source Code

```
DD9A 84FD STY FLPTR+1
DD9C A085 FLD1P LDY #FPREC-1 ; #BYTES ENTER W/FLPTR SET
DD9E B1FC FLD11 LDA [FLPTR],Y ; MOVE
DDA0 990000 STA FR1,Y
DDA3 88 DEY
DDA4 10F8  "DDA9 84FD STY FLPTR+l
DDA6 60 RTS

* STORE FR0 IN [X,Y] OR [FLPTR]

DDA7 86FC FSTOR STX FLPTR
DDA9 84FD STY FLPTR+1
DDAB A085 FSTOR LDY #FPREC-1 ; ENTRY W/FLPTR SET
DDAD 990400 FSTOR LDA FR0,Y
DDB0 91FC STA [FLPTR],Y
DDB2 88 DEY
DDB3 10F8  "DDB7 86FC FSTOR STX FLPTR
DDB5 60 RTS

* MOVE FR0 TO FR1

DDB6 A205 FMOVE LDX #FPREC-1
DDB8 B5D4 FMOVE1 LDA FR0,X
DDBA 95E0 STA FR1,X
DDBC CA DEX
DDBD 10F9  "DDBE A5D4 LDA FR0
DDBF 85F1 STA XFMFLG CLEAR TRANSFORM FLAG

EXP[X] and EXP10[X]

DDC2 A0DE LDX #LOG10E&$FF ; E**X = 10**[X*LOG10E]
DDC4 2090DD JSR FLD1R
DDC6 20BDAA JSR FMUL
DDC8 A07F  "DE4B BCS EXPERR
DDC9 A900 LDA EXP10 LDA #0 ; 10**X
DDCE 85F1 STA XFMFLG ; CLEAR TRANSFORM FLAG
DDD0 A5D4 LDA FR0
DDD2 85F0 STA SGNFLG ; REMEMBER ARG SGN
DDD4 29F7 ADD #$7F ; & MAKE PLUS
DDD6 85D4 STA FR0
DDDA 38 SEC
DDDB 9400 SBC #$40
DDDB 3026  "DE03 BMI EXP1

* 10**X = 10**[I+F] = [10**I] * [10**F]

DDDC C904 CMP #FPREC-2
DDDE 106A  "DE4B BPL EXPERR ; ARG TOO BIG
DDDF A266 LDX #FPSCR&$FF
DDDE A085 LDX #FPSCR/$100
DDD8 20D209 JSR FPI
DDD9 A5D4 LDA FR0
DDDB 85F1 STA XFMFLG ; SAVE MULTIPLIER EXP IN XFORM
DDDC 85D4 STA FR0 ; MAKE INTEGER
DDDE A085 LDX #FPSCR&$FF
DDD8 20D209 JSR FPI ; GET ARG BACK
DDDF 20BDAA JSR FSUB ; ARG - INTEGER PART = FRACTION
DDDH 20BD00 JSR FMUL
DDDI A266 LDX #FPSCR&$FF
DDDF A085 LDX #FPSCR/$100
DDDG 20BD00 JSR FD0R
DE00 20BD0A JSR FSUB ; ARG - INTEGER PART = FRACTION
* NOW HAVE FRACTION PART OF ARG [F] IN FR0,
* INTEGER PART [I] IN XFORM.
* IN XFMFLG. USE SERIES APPROX FOR
* 10**F, THEN MULTIPLY BY 10**I
DE02 C904 EXP1
DE03 A90A LDA #NPCOEF
DE05 A24D LDX #P10COEF&$FF
DE07 A0DE LDX #P10COEF/$100
```
Source Code

; [X-C][X+C]
Z = [X-C][X+C]

DE09 2040DD JSR PLYEVL ; P[X]
DE0C 20B6DD JSR PMOVE ; P[X]*P[X]
DE0D 20BDAA JSR FMUL ; DID WE TRANSFORM ARG
DE12 A5F1 LDA XMPLF ; NO SO LEAVE RESULT ALONE
DE14 F923 DE39 BEQ EXPSON ; NO SO LEAVE RESULT ALONE
DE16 1B CLC ; I/2
DE17 +A6A ROR A ; SAVE AS EXP-TO-BE
DE1A A901 LDA $1 ; GET MANTISSA BYTE
DE1C 9002 DE20 BCC EXP2 ; CHECK BIT SHIFTED OUT OF A
DE1E A910 LDA #$10 ; I WAS ODD - MANTISSA = 10
DE20 B5E1 EXP2 STA FR1+1
DE22 A204 LDX #$FREC-2
DE24 A900 LDA $0 ; CLEAR REST OF MANTISSA
DE26 95E2 EXP3 STA FR1+2,X
DE28 CA DEX
DE29 10FB DE26 BPL EXP3
DE2B A5E0 LDA FR1 ; BACK TO EXPONENT
DE2D 18 CLC
DE2E 6940 ADC #$40 ; BIAS IT
DE30 B019 DE4B BCS EXPERR ; OOPS... IT'S TOO BIG
DE32 3017 DE4B BMI EXPERR
DE34 85E8 STA FR1
DE36 20BDAA JSR FMUL ; FR1 = 10**I
DE39 A5F0 EXPSON LDA SGNFLG ; WAS ARG<0
DE3B 100D DE4A BPL EXPOUT ; NO-DONE
DE3D 20B6DD JSR PMOVE ; YES-INVERT RESULT
DE40 A2BF LDX #FONE&$FF
DE42 A6DE LDY #FONE/$100
DE44 2089DD JSR PLDR
DE47 2028DB JSR FDIV
DE4A 60 EXPOUT RTS ; [PANT, PANT - FINISHED:] EXPERR SEC ; FLAG ERROR
DE4C 60 RTS
& QUIT
DE4D 3D17941900 P10COF .BYTE $3D,$17,$94,$19,0,0 ;0.0000179419
00
DE53 3D57330500 .BYTE $3D,$57,$33,$05,0,0 ;0.0000573305
00
DE59 3E05547662 .BYTE $3E,$05,$54,$76,$62,0 ;0.0005547662
00
DE5F 3E32196227 .BYTE $3E,$32,$19,$62,$27,0 ;0.0032176227
00
DE65 3F016B6030 .BYTE $3F,$01,$6B,$60,$30,$36 ;0.016B603036
36
DE6B 3F07320327 .BYTE $3F,$07,$32,$03,$27,$41 ;0.0732032741
41
DE71 3F25433456 .BYTE $3F,$25,$43,$34,$56,$75 ;0.2543345675
75
DE77 3F66273730 .BYTE $3F,$66,$27,$37,$30,$50 ;0.6627373050
50
DE7D 4001151292 .BYTE $40,$01,$15,$12,$92,$55 ;1.15129255
55
DE83 3F99999999 .BYTE $3F,$99,$99,$99,$99,$99 ;0.999999999
99
= 000A NPCOEF EQU (*-P10COF)/FPREC
DE89 3F43429448 LOG10E .BYTE $3F,$43,$42,$94,$48,$19 ; LOG10[E]
19
DE8F 4001000000 FONE .BYTE $40,1,0,0,0,0 ; 1.0
00

z = [x-c][x+c]
DEA4 2098DD JSR FLDIR ; X+C
DEA7 2066DA JSR FADD
DEAA A0E6 LDX #FPSCR&$FF
DEAC A005 LDY #FPSCR/$100
DEAE 207DD JSR FST0R
DEB1 A2E0 LDX #PLYARG&$FF
DEB3 A005 LDY #PLYARG/$100
DEB5 2089DD JSR FLD0R
DEB8 A6FE LDX FPTR2
DEBA A4FF LDY FPTR2+1
DEBC 2099DD JSR FLDlR
DEBF 2060DA JSR FSUB ,X-C
DEC2 A2E6 LDX #FPSCR&$FF
DEC4 A005 LDY #FPSCR/$100
DEC6 2098DD JSR FLD1R
DEC9 202DB JSR FDIV ; [X-C]/[X+C] = Z
DECC 60 RTS

LOG10[X]

DECD A901 LOG LDA #1 ; REMEMBER ENTRY POINT
DECF 0002 "DED3 BNE LOGBTH
DED1 A900 LOG10 LDA #0 ; CLEAR FLAG
DED3 85F0 LOGBTH STA SGNFLG ; USE SGNFLG FOR LOG/LOG10 MARKER
DED5 A504 LDA FR0
DED7 1002 "DEDB BPL LOG5
DED9 38 LOGERR SEC
DEDA 60 RTS
DEDB LOG5
* WE WANT X = F*[llil**Y), 1<F<10
* 10**Y HAS SAME EXP BYTE AS X & MANTISSA BYTE = 1 OR 10
DED8 A5D4 LOG1 LDA FR0
DEDD 85E0 STA FR1
DEDF 38 SEC
DEEG E940 SBC #$40
DEEH ASLA A
DEEI 85F1 STA XFMPLG ; REMEMBER Y
DEE5 A5D5 LDA FR0+1
DEE7 29F0 AND #$F0
DEE9 D004 "DEEF BNE LOG2
DEEB A901 LDA #1
DEED D004 "DEF3 BNE LOG3
DEEF 85F1 LOG2 INC XFMPLG ; BUMP Y
DEF1 A910 LDA #$10
DEF3 85E1 LOG3 STA FR1+1 ; SET UP MANTISSA
DEF5 A204 LDX #FPREC-2 ; CLEAR REST OF MANTISSA
DEF7 A900 LDA #0
DEPF 95E2 LOG4 STA FR1+2,X
DEPB CA DEX
DEFC 10FB "DEF9 BPL LOG4
DEFE 202DB JSR FDIV ; X = X/[10**Y] - S.B. IN [1,10] ; ;LOG10[X],1<X<10

270
Source Code

DF25 20DBDA JSR FMUL ; Z*P[Z*Z]
DF28 A26C LDX #HALF&$FF
DF2A A0DF LDY #HALF/$100
DF2C 2099DD JSR FLDIR
DF2F 2066DA JSR PADD ; 0.5 + Z*P[Z*Z]
DF32 2066DD JSR PMOVE
DF35 A900 LDA #0
DF37 85D5 STA FR0+1
DF39 A5F1 STA XFMR
DF3B 85D4 STA FR0
DF3D 1007 'DF46 BPL LOG6
DF3F 49FF EOR #-1 ; FLIP SIGN
DF41 16 CLC
DF42 6901 ADC #1
DF44 85D4 STA FR0
DF46 LOG6 JSR IFP ; LEAVES FR1 ALONE
DF49 24F1 BIT XFMFLG
DF4B 1006 'DF53 BPL LOG7
DF4D A900 LDA #$80
DF4F 05D4 ORA FR0 ; FLIP AGAIN
DF51 85D4 STA FR0
DF53 LOG7 JSR PADD ; LOG[X] = LOG[X] +Y
DF55 2066DA LOGOUT
DF56 483162277 SQR10 .BYTE $40,$03,$16,$22,$77,$66 ; SQUARE ROOT OF 10
DF56 3F50000000 FHALF .BYTE $3F,$50,0,0,0,0 ; 0.5
DF58 A900 BEQ LOGDON ; WAS LOG10, NOT LOG
DF5A A289 LDX #LOG10E&255 ; LOG[X]/LOG10[E]
DF5C A0DE LDY #LOG10E/$100
DF5E 2099DD JSR FLDIR
DF61 2028DB JSR FDIV
DF64 18 LOGDON CLC
DF65 60 RTS
DF66 483162277 SQR10 .BYTE $40,$03,$16,$22,$77,$66 ; SQUARE ROOT OF 10
DF66 3F50000000 FHALF .BYTE $3F,$50,0,0,0,0 ; 0.5
DF72 3F49155711 LGCOEF .BYTE $3F,$49,$15,$57,$11,$08 ; 0.4915571108
DF78 85F1704947 .BYTE $BF,$51,$70,$49,$47,$08 ;-0.5170494708
DF7E 3F39205761 .BYTE $3F,$39,$20,$57,$61,$95 ; #3920576195
DF84 BF04396303 .BYTE $BF,$04,$39,$63,$03,$55 ; #BF04396355
DF8A 3F10093012 .BYTE $3F,$10,$09,$30,$12,$64 ; 0.1009301264
DF90 3F09390004 .BYTE $3F,$09,$39,$00,$84,$60 ; 0.0939008460
DF96 3F12425847 .BYTE $3F,$12,$42,$50,$47,$42 ; 0.1242584742
DF9C 3F17371286 .BYTE $3F,$17,$37,$12,$06,$08 ; 0.1737128608
DFA2 3F28952971 .BYTE $3F,$28,$95,$29,$71,$17 ; 0.2895297117
DFA8 3F36858986 .BYTE $3F,$36,$85,$88,$96,$44 ; 0.3685898644
DFE4 3E16504449 NLCOEF EQU (*-LGCOEF)/FPREC
DFE8 3E16504449 ATCOEF .BYTE $3E,$16,$05,$44,$94,0 ; 0.0016504449
DFB4 BE9563845 .BYTE $BE,$95,$68,$38,$45,0 ; -0.09563845
DFBA 3F2687994 .BYTE $3F,$02,$68,$79,$94,$16 ; 0.0268799416
DFC8 BF04927890 .BYTE $BF,$04,$92,$78,$90,$80 ; -0.0492789080
DFCC BF08922912 .BYTE $BF,$08,$92,$29,$12,$44 ; -0.0892291244
DFD2 3F11084809 .BYTE $3F,$11,$08,$40,$09,$11 ; 0.1108480911

271
DFDB  BF14283156   .BYTE  $BF, $14, $28, $31, $56, $04 ; -0.1428315604
DFDE  3F19999877   .BYTE  $3F, $19, $99, $98, $77, $44 ; 0.1999987744
DFE4  BF33333331   .BYTE  $BF, $33, $33, $33, $31, $13 ; -0.3333333113
DFEA  3F99999999   FP9S  .BYTE  $3F, $99, $99, $99, $99, $99 ; 0.999999999
       = $00006FATCF  EQU (* - ATCOEF) / FPREC
DFF0  3F78539816   PIOV4 .BYTE  $3F, $78, $53, $98, $16, $34 ; PI/4 = ARCTAN[1.0]

Atari Cartridge Vectors

DFF6  = BFF9  SCVECT CRTGI
BFF9  60  RTS
BFFA  00A0  DW COLDSTART ; COLDSTART ADDR
BFFC  00  DB 0 ; CART EXISTS
BFFD  05  DB 5 ; FLAG
BFFE  F9BF  DW SCVECT ; COLDSTART ENTRY ADDR

End of BASIC

C000  END
Macros in Source Code

The following is a listing of the macros used in this source listing. You will be able to tell when a macro was used by a plus (+) sign to the left of the hex code produced in column two by the assembler.

```
ASLA: MACRO
  %L ASL
ENDM
RORA: MACRO
  %L ROR
ENDM
LSRA: MACRO
  %L LSR
ENDM
ROLLA: MACRO
  %L ROL
ENDM
FDB: MACRO
  %L DW REV (%1)
  IF '"=%2' <> '=
    DW REV (%2)
  IF '"=%3' <> '=
    DW REV (%3)
  IF '"=%4' <> '=
    DW REV (%4)
  IF '"=%5' <> '=
    DW REV (%5)
  ENDIF
ENDIF
ENDIF
ENDM
LOCAL: MACRO
PROC
ENDM
BYTE: MACRO
  %L DB $80+(((%2-*)&$7F) XOR $40)
ENDM
```

Syntax Table Macro

: THIS MACRO IS USED TO SIMULATE THE ACTION OF THE ORIGINAL
: ASSEMBLER IN HANDLING SPECIAL SYNTAX TABLE PSEUDO OPS AND
: OPERANDS
:
: THE 'SYN' MACRO EXAMINES UP TO 4 ARGUMENTS FOR CERTAIN SPECIAL
: CASE NAMES.
:
: IF THE NAME 'JS' IS FOUND, IT GENERATES A SPECIAL 'RELATIVE
: SYNTAX JSR' TO THE LABEL FOUND IN THE NEXT PARAMETER
; IF THE NAME 'AD' IS FOUND, IT GENERATES A WORD ADDRESS OF
; THE LABEL FOUND IN THE NEXT PARAMETER
;
; ANY OTHER NAME IS ASSUMED TO BE A SIMPLE BYTE VALUE

SYN: MACRO
:SYAR2 SET '='%2<>'='
:SYAR3 SET '='%3<>'='
:SYAR4 SET '='%4<>'='
IF '='%1'=' 'JS'
%L DB $80+((%2-*$7F) XOR $40 )
:SYAR2 SET 0
ELSE
  IF '='%1'=' 'AD'
  %L DW (%2)
  :SYAR2 SET 0
  ELSE
  %L DB %1
  ENDIR
ENDIF

IF :SYAR2
IF '='%2'=' 'JS'
DB $80+((%3-*$7F) XOR $40 )
:SYAR3 SET 0
ELSE
  IF '='%2'=' 'AD'
  DW (%3)
  :SYAR3 SET 0
  ELSE
  DB %2
  ENDIR
ENDIF
ENDIF

IF :SYAR3
IF '='%3'=' 'JS'
DB $80+((%4-*$7F) XOR $40 )
:SYAR4 SET 0
ELSE
  IF '='%3'=' 'AD'
  DW (%4)
  :SYAR4 SET 0
  ELSE
  DB %3
  ENDIR
ENDIF
ENDIF

IF :SYAR4
IF '='%4'=' 'JS'
DB $80+((%5-*$7F) XOR $40 )
ELSE
  IF '='%4'=' 'AD'
  DW (%5)
  ELSE
  DB %4
  ENDIR
ENDIF
ENDIF
ENDIF

ENDM
Appendix B

The Bugs in Atari BASIC

Yes, it’s true. There are some bugs in Atari BASIC. Of course, that’s not surprising, since Atari released the product as ROM without giving the authors a chance to do second-round bug-fixing. But what hurts, a little, is that most of the fixes for the bugs have been known since June of 1979.

As this book is being written, rumor has it that at last Atari is in the final stages of releasing a new version of the BASIC ROMs. Unfortunately, these modified ROMs will appear too late for us to comment upon them in this edition. On the other hand, there are supposed to be fewer than twenty fixes implemented (which isn’t a bad record for a product as mature as Atari BASIC), so those of you who are willing to PEEK around a bit can use this listing as at least a road map to the new ROMs.

In any case, though, we thought it would be appropriate to mention a few of the bugs we know about, show you why they exist, and tell how we fixed them back there in the summer of ’79.

The Editing and String Bug
In the course of editing a BASIC program, sometimes the system loses all or part of the program, or it simply hangs. Often, even SYSTEM RESET will not return control to the user.

Also, string assignments that involve the movement of exact multiples of 256 bytes do not move the bytes properly. For example, A$ = B$(257,512) would actually move bytes 513 through 768 of B$ into bytes 257 through 512 of A$, even if neither string were DIMensioned to those values.

Both of these are really the same bug. And both are caused because we strove to be a little too efficient.

There are many ways to move strings of bytes using the 6502’s instruction set. The simplest and most-used methods, though, are excruciatingly slow. So Paul and Kathleen invented a super-fast set of move-memory routines, one for
moving up in memory (EXPAND, at $A881) and one for moving down in memory (CONTRACT, at $A8FD).
Unfortunately, the routines are very complex (which is what makes them fast) and difficult to interface with properly. And so a bug crept into CONTRACT.
Take a look at the code of FMOVER ($A947). When we get here, we expect MVLNG to contain the complement of the least significant byte of the move length while MVLNG + 1 contains its most significant byte. But look what happens if the original move length was, for example, $200. The complement of the least significant byte ($00) is still zero ($00), so the BEQ to :CONT4 occurs immediately.
But by then, the X register contains the number of pages to move plus one (X would contain 3 in this example), so we increment it (it becomes 2) and go to label :CONT3, where we bump the high-order byte of both the source and destination addresses. Ah, but therein lies the rub! We haven’t yet done anything with the first values in those source and destination addresses, so we have effectively skipped 256 bytes of each!
The solution is to replace the BEQ :CONT4 at $A94E with the following code:

DEX
BNE :CONT2
RTS
Do you see the difference? If we enter with MVLNG equal to zero, we immediately move 256 bytes (at :CONT2) before ever attempting to change the source and destination addresses.
And this fix works, honest. We’ve been using it like this for over two years in BASIC A+.

Minus Zero
Taking the unary minus of a number (A = 0: PRINT -A) can result in garbage. Usually, this garbage will not affect subsequent calculations, but it does print strangely. And how did this come about?
We simply forgot to take into consideration the fact that zero doesn’t really have a sign. Look at the code for the unary minus operator (XPUMINUS, at $ACA8). Do you see the problem? We simply invert the most significant bit (the sign bit) of the floating point number in FR0.
What we should have coded would be something like this:

LDA   FR0
BEQ   :NOINVERT
EOR  #$80
STA   FR0
:NOINVERT

Luckily, this is not too severe a problem to the BASIC user (one can always use "PRINT 0-A" instead of "PRINT -A"), but just think — it only cost two bytes to fix this bug.

**LOCATE and GET**

The GET statement does not reinitialize its buffer pointer, so it can do nasty things to memory if used directly after a statement which has changed the system buffer pointer. For example, GET can change the line number of a DATA statement if it is used after a READ. Also, the same problem exists for the LOCATE statement, since it calls GET.

From BASIC, the easiest solution is to use a function or statement which is known to reset the pointer. Coding "XX=STR$(0)" works just fine, as does PRINTing any number.

Within the source listing, the problem exists at location $BC82, label GET1. If the code had simply read as follows, there would be no bug:

GET1
    JSR INTLBF ; reset buffer pointer
    LDA #ICGTC ; continue as before

**INPUT and READ**

Using either an INPUT or READ statement without a following variable does not cause a syntax error (as it should). Then, attempting to execute a statement such as 20 INPUT can cause total system lock-up.

The solution from BASIC? Be careful and don’t do it.

And this is one bug that we will not show the fix for, simply because it’s too long and involved. We will, however, point to labels :SINPUT and :SREAD (at locations $A6F4 and $A6F5) in the Syntax Tables and show why the bug exists.

Note that the :SINPUT does a syntax call (SYNS,) to the :OPD syntax, which looks for — but does not insist upon — a file number specifier (#<numeric expression>). Then the
syntax joins with :SREAD, which looks for zero or more variables.

Oops! Zero or more? Shouldn’t that be one or more? That’s where the problem lies.

**Do Not Use NOT**

In all too many cases, the use of the NOT operator is guaranteed to get you in trouble. If you don’t believe it, try this: PRINT NOT NOT 1.

The explanation of why the bug occurs is too lengthy to give in detail here; suffice it to say that the precedence of NOT is wrong. Remember the Operator Precedence Table we displayed in Chapter 8 of Part 2? Look at what you got for the go-onto-stack and come-off-stack precedence values for NOT.

Or look at location $AC57, the NOT entry in OPRTAB. NOT uses a 7 for both its precedence values. But wait a minute. If two operators have the same apparent precedence (as in NOT NOT A or even A + B + C), the expression executor will pop the first one off the stack and execute it. But with a unary operator, there is nothing to execute yet.

And the same bug exists for both unary minus and unary plus, so −−3 and ++5 don’t execute properly. Of course, since unary plus doesn’t really do anything, it doesn’t matter. In the case of unary minus, though, all but the last minus sign in a string of minus signs is ignored (that is, −−3 produces −3 as a result, instead of +3, as it should). But, by an incredible coincidence, the damage that unary minus causes is invisible to Execute Expression as a whole and only produces the error noted.

The fix? Well, if we want to leave NOT where it is in the order of things, the only way is to restructure the whole precedence table. But if we are willing to accord it a very high precedence, like unary plus and minus, we can fix it — and plus and minus — by changing the bytes at $AC57, $AC64, and $AC65 to $DC. And, thanks to the differing go-onto-stack and come-off-stack values, we can stack as many NOTs, pluses, or minuses as we want.

Are these all the bugs we know about that can be fixed easily? No. But these are the easiest to understand or the easiest to fix, and we thought they were instructive.

Of course, unless you have an EPROM board and burner handy, you may not be able to take advantage of these fixes.
But at least now you may be able to work around them as you program with good old buggy-version Atari BASIC.

And take heart. Remember Richard’s Rule: Any nontrivial piece of software has bugs in it. And the corollary: Any piece of software which is bug-free is trivial.
Appendix C

Labels and Hexadecimal Addresses

<table>
<thead>
<tr>
<th>Label</th>
<th>Hexadecimal Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>AADD</td>
<td>AF52</td>
</tr>
<tr>
<td>AAPSTR</td>
<td>AB98</td>
</tr>
<tr>
<td>n ADC</td>
<td>AF53</td>
</tr>
<tr>
<td>ADFLAG</td>
<td>0031</td>
</tr>
<tr>
<td>n AMUL1</td>
<td>AF5D</td>
</tr>
<tr>
<td>AMUL2</td>
<td>AF46</td>
</tr>
<tr>
<td>APHM</td>
<td>003E</td>
</tr>
<tr>
<td>ARGOPS</td>
<td>0080</td>
</tr>
<tr>
<td>ARGPOP</td>
<td>0082</td>
</tr>
<tr>
<td>ARGPO2</td>
<td>AC26</td>
</tr>
<tr>
<td>ARGPO4</td>
<td>ABF2</td>
</tr>
<tr>
<td>ARGPO8</td>
<td>ABB8</td>
</tr>
<tr>
<td>ARGSTK</td>
<td>008A</td>
</tr>
<tr>
<td>n ASCIN</td>
<td>D800</td>
</tr>
<tr>
<td>ASLA</td>
<td>mac</td>
</tr>
<tr>
<td>ATAN</td>
<td>B77</td>
</tr>
<tr>
<td>ATAN1</td>
<td>BE9A</td>
</tr>
<tr>
<td>ATAN2</td>
<td>B6D4</td>
</tr>
<tr>
<td>ATCOEF</td>
<td>DFAE</td>
</tr>
<tr>
<td>n ATTEMP</td>
<td>00AF</td>
</tr>
<tr>
<td>ATNOUT</td>
<td>BEE2</td>
</tr>
<tr>
<td>n BOTH</td>
<td>BD3</td>
</tr>
<tr>
<td>BRKBYT</td>
<td>0011</td>
</tr>
<tr>
<td>BYTE</td>
<td>C044</td>
</tr>
<tr>
<td>n BYTE2</td>
<td>C044</td>
</tr>
<tr>
<td>CAASN</td>
<td>002D</td>
</tr>
<tr>
<td>CACOM</td>
<td>003C</td>
</tr>
<tr>
<td>CADR</td>
<td>0043</td>
</tr>
<tr>
<td>CALPRN</td>
<td>0030</td>
</tr>
<tr>
<td>n CAND</td>
<td>002A</td>
</tr>
<tr>
<td>CASC</td>
<td>0040</td>
</tr>
<tr>
<td>CHCR</td>
<td>003E</td>
</tr>
<tr>
<td>n COMM</td>
<td>0012</td>
</tr>
<tr>
<td>CCR</td>
<td>0016</td>
</tr>
<tr>
<td>CDATA</td>
<td>D001</td>
</tr>
<tr>
<td>CDIV</td>
<td>0012</td>
</tr>
<tr>
<td>n CDLPR1</td>
<td>0039</td>
</tr>
<tr>
<td>n CDOL</td>
<td>0013</td>
</tr>
<tr>
<td>n CDO</td>
<td>001D</td>
</tr>
<tr>
<td>n CDSLPR</td>
<td>003B</td>
</tr>
<tr>
<td>n CEOS</td>
<td>0014</td>
</tr>
<tr>
<td>n CER</td>
<td>001C</td>
</tr>
<tr>
<td>n CEXP</td>
<td>0023</td>
</tr>
<tr>
<td>CFFUN</td>
<td>003D</td>
</tr>
<tr>
<td>CFLPRN</td>
<td>003A</td>
</tr>
<tr>
<td>n CGR</td>
<td>0089</td>
</tr>
<tr>
<td>n CGE</td>
<td>001F</td>
</tr>
<tr>
<td>CGSUB</td>
<td>008C</td>
</tr>
<tr>
<td>CGS</td>
<td>0018</td>
</tr>
<tr>
<td>CGT</td>
<td>0021</td>
</tr>
<tr>
<td>n CTO</td>
<td>0017</td>
</tr>
<tr>
<td>CILET</td>
<td>0036</td>
</tr>
<tr>
<td>CIO</td>
<td>E456</td>
</tr>
<tr>
<td>n CIX</td>
<td>F0P2</td>
</tr>
<tr>
<td>CLALL1</td>
<td>0D4F</td>
</tr>
<tr>
<td>n CLE</td>
<td>001D</td>
</tr>
<tr>
<td>CLEN</td>
<td>0042</td>
</tr>
<tr>
<td>CLIST</td>
<td>0004</td>
</tr>
<tr>
<td>n CLPRN</td>
<td>002B</td>
</tr>
<tr>
<td>n CLSALL1</td>
<td>BD41</td>
</tr>
<tr>
<td>CLSYS1</td>
<td>BC51</td>
</tr>
<tr>
<td>n CLSYSVD</td>
<td>BC7F</td>
</tr>
<tr>
<td>n CLT</td>
<td>0020</td>
</tr>
<tr>
<td>n CMINUS</td>
<td>0026</td>
</tr>
<tr>
<td>n COLOR</td>
<td>00CB</td>
</tr>
<tr>
<td>n COMCNT</td>
<td>00B0</td>
</tr>
<tr>
<td>n CON</td>
<td>001E</td>
</tr>
<tr>
<td>CONTLO</td>
<td>0AE8</td>
</tr>
<tr>
<td>n CONTRA</td>
<td>0AFD</td>
</tr>
<tr>
<td>n COPEN</td>
<td>0BB6</td>
</tr>
<tr>
<td>n COR</td>
<td>0029</td>
</tr>
<tr>
<td>n COS</td>
<td>0D11</td>
</tr>
<tr>
<td>n COX</td>
<td>0094</td>
</tr>
<tr>
<td>n CPC</td>
<td>009D</td>
</tr>
<tr>
<td>CPLUS</td>
<td>0025</td>
</tr>
<tr>
<td>n CPND</td>
<td>001C</td>
</tr>
<tr>
<td>n CR</td>
<td>009B</td>
</tr>
<tr>
<td>n CREAD</td>
<td>0022</td>
</tr>
<tr>
<td>n CREDS</td>
<td>02C4</td>
</tr>
<tr>
<td>n CRPRN</td>
<td>002C</td>
</tr>
<tr>
<td>n CRTGI</td>
<td>BFF9</td>
</tr>
<tr>
<td>n CSASN</td>
<td>002E</td>
</tr>
<tr>
<td>n CSC</td>
<td>0015</td>
</tr>
<tr>
<td>n CSE3</td>
<td>034</td>
</tr>
<tr>
<td>n CSGB</td>
<td>0031</td>
</tr>
<tr>
<td>n CSGL</td>
<td>033</td>
</tr>
<tr>
<td>n CSIE</td>
<td>02F</td>
</tr>
<tr>
<td>n CSLRN</td>
<td>037</td>
</tr>
<tr>
<td>n CSLT</td>
<td>032</td>
</tr>
<tr>
<td>n CSNE</td>
<td>030</td>
</tr>
<tr>
<td>n CSOE</td>
<td>0011</td>
</tr>
<tr>
<td>n CSROP</td>
<td>01D</td>
</tr>
<tr>
<td>n CSTEP</td>
<td>01A</td>
</tr>
<tr>
<td>n CSTR</td>
<td>03D</td>
</tr>
<tr>
<td>n CTEN</td>
<td>01B</td>
</tr>
<tr>
<td>n CTO</td>
<td>019</td>
</tr>
<tr>
<td>n CUMINU</td>
<td>036</td>
</tr>
<tr>
<td>n CUPLUS</td>
<td>0035</td>
</tr>
<tr>
<td>n CUSR</td>
<td>03F</td>
</tr>
<tr>
<td>n CVFFP</td>
<td>D800</td>
</tr>
<tr>
<td>n CVVAL</td>
<td>0041</td>
</tr>
<tr>
<td>n CVEFASC</td>
<td>D8E6</td>
</tr>
<tr>
<td>n CVFPI</td>
<td>AD56</td>
</tr>
<tr>
<td>n CVIPF</td>
<td>D9A</td>
</tr>
<tr>
<td>DATAD</td>
<td>0086</td>
</tr>
<tr>
<td>n DATALM</td>
<td>0087</td>
</tr>
<tr>
<td>n DEGFLG</td>
<td>00FB</td>
</tr>
<tr>
<td>n DEGON</td>
<td>0086</td>
</tr>
<tr>
<td>n DIGRT</td>
<td>00F1</td>
</tr>
<tr>
<td>n DIRFLG</td>
<td>0A6</td>
</tr>
<tr>
<td>n DMESR</td>
<td>BC58</td>
</tr>
<tr>
<td>n DOSLOC</td>
<td>008A</td>
</tr>
<tr>
<td>n DSPLG</td>
<td>02FE</td>
</tr>
<tr>
<td>n ECSRIZE</td>
<td>00A4</td>
</tr>
<tr>
<td>n EEEX</td>
<td>00ED</td>
</tr>
<tr>
<td>n EELAVC</td>
<td>BABD</td>
</tr>
<tr>
<td>n ENDSTA</td>
<td>00B8</td>
</tr>
<tr>
<td>n ENDVOL</td>
<td>0088</td>
</tr>
<tr>
<td>n ENTD</td>
<td>0084</td>
</tr>
<tr>
<td>n ECH</td>
<td>0550</td>
</tr>
<tr>
<td>n EBRMT</td>
<td>B29</td>
</tr>
<tr>
<td>n EGRG</td>
<td>B22</td>
</tr>
<tr>
<td>n ERLTL</td>
<td>B24</td>
</tr>
<tr>
<td>n EROF</td>
<td>B26</td>
</tr>
<tr>
<td>n EROML</td>
<td>B28</td>
</tr>
<tr>
<td>n EROV</td>
<td>B2A</td>
</tr>
<tr>
<td>n EROFL</td>
<td>B2A</td>
</tr>
<tr>
<td>n ERRAO</td>
<td>B2C</td>
</tr>
<tr>
<td>n ERRDIM</td>
<td>B2E</td>
</tr>
<tr>
<td>n ERRD</td>
<td>B2E</td>
</tr>
<tr>
<td>n ERRNO</td>
<td>B18</td>
</tr>
<tr>
<td>n ERRRP</td>
<td>B10</td>
</tr>
<tr>
<td>n ERRRL</td>
<td>B12</td>
</tr>
<tr>
<td>n ERRNSP</td>
<td>B16</td>
</tr>
<tr>
<td>n ERRNUM</td>
<td>00B9</td>
</tr>
<tr>
<td>n ERRORD</td>
<td>B34</td>
</tr>
<tr>
<td>n ERROR</td>
<td>B94</td>
</tr>
<tr>
<td>n ERRR</td>
<td>B94</td>
</tr>
<tr>
<td>n ERRTP</td>
<td>B1A</td>
</tr>
<tr>
<td>n ERRSAV</td>
<td>00C3</td>
</tr>
<tr>
<td>n ERRSSL</td>
<td>B93</td>
</tr>
<tr>
<td>n ERRUSF</td>
<td>B38</td>
</tr>
<tr>
<td>n ERVAL</td>
<td>B91</td>
</tr>
<tr>
<td>n ERVAL1</td>
<td>B9A</td>
</tr>
<tr>
<td>n ESIGN</td>
<td>00EF</td>
</tr>
<tr>
<td>n EVAADR</td>
<td>0022</td>
</tr>
<tr>
<td>n EVADI</td>
<td>0084</td>
</tr>
<tr>
<td>n EVAD2</td>
<td>0066</td>
</tr>
<tr>
<td>n EVARRA</td>
<td>0040</td>
</tr>
<tr>
<td>n EVDM</td>
<td>0001</td>
</tr>
<tr>
<td>n EVNUM</td>
<td>0001</td>
</tr>
<tr>
<td>n EVSADR</td>
<td>0002</td>
</tr>
<tr>
<td>n EVSCAL</td>
<td>0000</td>
</tr>
<tr>
<td>n EVSDIM</td>
<td>0006</td>
</tr>
<tr>
<td>n EVSDTA</td>
<td>0022</td>
</tr>
<tr>
<td>n EVSLEN</td>
<td>0048</td>
</tr>
<tr>
<td>n EVSTR</td>
<td>0008</td>
</tr>
<tr>
<td>n EVTYPE</td>
<td>0000</td>
</tr>
<tr>
<td>n EVVAL1</td>
<td>0022</td>
</tr>
<tr>
<td>n EXECLN</td>
<td>A95F</td>
</tr>
<tr>
<td>n EXECONS</td>
<td>A962</td>
</tr>
<tr>
<td>n EXEXPR1</td>
<td>AAE0</td>
</tr>
<tr>
<td>n EXOP</td>
<td>AB08</td>
</tr>
<tr>
<td>n EXP</td>
<td>DDC8</td>
</tr>
<tr>
<td>n EXP1</td>
<td>D683</td>
</tr>
<tr>
<td>n EXP2</td>
<td>D28</td>
</tr>
<tr>
<td>n EXP3</td>
<td>D26</td>
</tr>
<tr>
<td>n EXPAND</td>
<td>A881</td>
</tr>
<tr>
<td>n EXPERR</td>
<td>DB4</td>
</tr>
<tr>
<td>n EXPINT</td>
<td>ABE2</td>
</tr>
<tr>
<td>n EXLOW</td>
<td>A87F</td>
</tr>
<tr>
<td>n EXPOUT</td>
<td>D4A</td>
</tr>
<tr>
<td>n EXPSGN</td>
<td>D39</td>
</tr>
<tr>
<td>n EXSVOP</td>
<td>00AB</td>
</tr>
<tr>
<td>n EXSVPR</td>
<td>00AC</td>
</tr>
<tr>
<td>n FADD</td>
<td>DAA6</td>
</tr>
<tr>
<td>n n FASC</td>
<td>DBE6</td>
</tr>
<tr>
<td>n n FBODY</td>
<td>DB0C</td>
</tr>
<tr>
<td>n n FCHRFL</td>
<td>E08F</td>
</tr>
<tr>
<td>n n FDB</td>
<td>mac</td>
</tr>
<tr>
<td>n n FDIV</td>
<td>DB28</td>
</tr>
<tr>
<td>n n FHALF</td>
<td>DF6C</td>
</tr>
<tr>
<td>n n FIXST</td>
<td>BB25</td>
</tr>
<tr>
<td>n n FL0D</td>
<td>D808</td>
</tr>
<tr>
<td>n n FL0D2</td>
<td>D80D</td>
</tr>
<tr>
<td>n n FL0D8</td>
<td>D89</td>
</tr>
<tr>
<td>n n FLD01</td>
<td>D9C</td>
</tr>
<tr>
<td>n n FLD1R</td>
<td>D98</td>
</tr>
<tr>
<td>n n FLIM</td>
<td>0000</td>
</tr>
<tr>
<td>n n FLIST</td>
<td>BAD5</td>
</tr>
<tr>
<td>n n FLOG10</td>
<td>DF01</td>
</tr>
<tr>
<td>n n FLPTR</td>
<td>00FC</td>
</tr>
<tr>
<td>n n FMOVE</td>
<td>DDB6</td>
</tr>
<tr>
<td>n n MOVE</td>
<td>00BB</td>
</tr>
<tr>
<td>n n MOVER</td>
<td>A947</td>
</tr>
<tr>
<td>n n FMPREC</td>
<td>0005</td>
</tr>
<tr>
<td>n n FMUL</td>
<td>DABB</td>
</tr>
<tr>
<td>n n FTCAB</td>
<td>A82B</td>
</tr>
<tr>
<td>n n FONE</td>
<td>DEBF</td>
</tr>
<tr>
<td>n n FPR9S</td>
<td>DFEA</td>
</tr>
<tr>
<td>n n FPI</td>
<td>D9D2</td>
</tr>
<tr>
<td>n n FPONE</td>
<td>BE71</td>
</tr>
<tr>
<td>n n FPOR</td>
<td>DB8</td>
</tr>
<tr>
<td>n n FPR2</td>
<td>00FE</td>
</tr>
<tr>
<td>n n FROM</td>
<td>0005</td>
</tr>
<tr>
<td>n n FR1</td>
<td>00B0</td>
</tr>
<tr>
<td>n n FRAM</td>
<td>00E1</td>
</tr>
<tr>
<td>n n FR2</td>
<td>0066</td>
</tr>
<tr>
<td>n n FRA10</td>
<td>DDB1</td>
</tr>
<tr>
<td>n n FRA1E</td>
<td>DDB9</td>
</tr>
<tr>
<td>n n FRA20</td>
<td>DDB5</td>
</tr>
<tr>
<td>n n FRA2E</td>
<td>DDBF</td>
</tr>
<tr>
<td>n n FRADD</td>
<td>AD3B</td>
</tr>
</tbody>
</table>
Symbol Index

Symbols

' ' in Operator Name Table 177
with string literals 130
(See also XPACOM)

, in Operator Name Table 177
precedence of 69-70
with array, in ONT 180
with PRINT 98

$ in hexadecimal 115
in Operator Name Table 177
in variable names 15, 46
(See alphabetic entry for terms
that begin with '...', like
:LPSCAN) 58
in Operator Name Table 177
with PRINT 98

: in Operator Name Table 177
with PRINT 98

# in Operator Name Table 178
with PRINT 98

< = (See also XPLE, XPSLE) 178-79
< > (See also XPNE, XPSNE) 178-79
>= (See also XPGE, XPSGE) 178-79
< (See also XPLT, XPSLT)
in ABML 34-39
in Operator Name Table 178-79
precedence of 56

> (See also XPGT, XPSGT)
in ABML 34-39
in Operator Name Table 178-79
precedence of 56

= (See also XPEQ, XPSEQ) 41-42,
58-64
in Operator Name Table 178-79
precedence of 55-56

& in Operator Name Table 178
precedence of 55-56, 58-64

(See also XPMUL, FMUL, FRMUL)
in Operator Name Table 178
precedence of 55-56, 58-64

+ (See also XPPLUS, XPUPLUS,
FADD, FRADD)
in Operator Name Table 178
unary 179, Appendix B
precedence of 55-56, 58-64

- (See also XPMINUS, XPUMINUS,
FSUB, FRSUB)
in Operator Name Table 178
unary 179, Appendix B

/ (See also XPDIV, FDIV, FRDIV)
178

(See also XPDLPNRN, XPALPRN,
XPSLPNRN)
in variable names 15, 46
mathematical, in Operator
Name Table 179
precedence of 69-70
string, array, DIM, and func-
tion, in ONT 179-80
tokens for 70

) (See also XPRPRN)
in Operator Name Table 179
precedence of 69-70

:: 34-37
i 34, 37, 41
= < 56
? as EOE operator 34-35, 58-64
95

Numbers

6502 microprocessor 1-2, 40

A
AADD 202, $AF52
ADR 66-68
AAPSTR 191, $AB98
ABS (See also XPABS) 69, 180, 206
Absolute Non-Terminal Vector (See
ANTV)
ABML (Atari BASIC Meta-Language)
33-34, 37
AD Appendix A
addition (See FADD, FRADD)
ADR 180
AMUL 202, $AF5D
AMUL 2 02, $AF46
AND (See also XPAND) 89, 179
ANTV (in ABML) 40-42, 44, 162
APHM 13, 143, $000E
application high memory 13
ARGOPS 23, 66, 143, $0080
ARGP2 192, $AC06
ARGPOP 192, $ABF2
ARGPUSH 65, 191, $ABBA
ARGSTK 23, 66, 143, $0080
arguments 56
Argument Stack 12, 23, 56-67
argument Stack 12, 23, 56-67
entry format 66-68
example of use 56-64
arithmetic assignment operator (See
XPAASN)
arithmetic expressions 12, 55-65
array variables 15-16, 18, 66-67, 69-70,
106-7, 127
Array/String Table (See String/Array Table)
ARSLVL 66, 144, $00AA
ARSTKX 144, $00AA
ASC (See also XPASC) 180, 204
ASCIN 246, $D800
ASLA: Appendix A
assembler 2
assembly language 2-3
ATAN[X] 244, $BE77
Atari BASIC
as a high-level language 2-5
location in memory 14
Meta-Language (ABML) 33
ROM pointer 143, $A000
Atari cartridge vectors 272
ATASCII 9, 47, 88, 90, 116, 135-36
AT LINE (in error message) 74, 231-32,
$89AE
ATN (See also XPATN, ATAN) 180, 207,
244
AUXn (i.e., AUX1, AUX2, etc.) 99-100
B
BASIC ROM pointer 143, $A000
binary 115, 119-20
blanks in program lines 27, 29
block move routines 20-23
BNF 33
BREAK 50, 96, 101
BRKBYT 101, 110, 143, $0011
buffer (See also INBUF, OUTBUF,
LBUF) 13, 65, 145
bugs Preface, 20-21, Appendix B
BYE (See also XBYE, :SBYE) 105
BYELOC 143, $E471
byte 119-20
BYTE: Appendix A
C
CALPRN 70
carriage return character 177
cartridge vectors 272
CDLPRN 70
CDSLPRN 70
CFLPRN 70
Change Last Token (in ABML) 41-42
CHNG (in ABML) 41-42, 45, 162
CHR$ (See also XCHR) 180, 205
CIO 25, 85, 91, 93, 102, 143, $E456
CIX 26, 29-30, 43, 45, 47
CLOAD (See also XCLOAD, :SCLOAD)
84, 237
CLOG 180
CLOSE (See also XCLOSE, :SCLOSE)
100, 146, 239
CLPRN 70
CLR (See also XCCLR, :SCLR) 83, 103, 224
CLSALL 242, $BD41
CLSYS 99, 241, $BCF1
CLOAD 241, $BCF1
COLDSTART 86, 101, 109-110, 147,
$A000
COLOR (See also XCOLOR, :SCOLOR)
execution 91, 233
memory location 91-92, 144,
$00C8
color registers (See also CREGS) 91-92,
143
COMMON (unused command; see
XCOM, :SCOM)
compiler 3-4
constants 33-34, 130
CONT (See also XCONT, :SCONT)
71-72, 225
:CONT2 183, Appendix B, $A954
:CONT3 183, Appendix B, $A950
:CONT4 183, Appendix B, $A95B
CONTOLOW 20-23, 31, 77, 182, $A8FB
CONTO 20-22, 28, 182-83, $A8FD, Appendix B
conversion
ASCII to floating point 145,
$00ED-$00F1
decimal to hexadecimal 116-17
floating point to ASCII (See also
CVFASC) 250-52
floating point to integer (See also
CVPFL, FPI) 197, 253-55
hexadecimal to decimal 116
integer to floating point (See also
CVFIP) 252-53
COPEN 237, $BBB6
COS (See also XPCOS, COS[X]) 105, 180,
207, 243
COS[X] 243, $BD61
COX 26, 27-30, 43, 45, 48, 97-98, 144,
$0094
CPC 43-44, 144, 153, $009D

286
CPU stack 43, 51, 74-75, 109-110
CREGS 143, $02C4
CRTG1 143, $BBF9
CSAVE (See also XCSAVE, :SCSAVE) 84, 237
CSLPRN 70
Current Program Counter (See CPC)
CVAFP 96, 246, $D800
CVFASC 98, 250, $D8E6
CVFPI (See also FPI) 197, $AD56
CVIFP 252, $D9AA
:CVFR0 263, $DC70

D

D1 169, $A705
DATA (See also XDATA, :SDATA) 103-104, 110, 131, 140
DATAD 103-4, 110, 144, $00B6
DATALN 103-4, 110, 144, $00B7
DCBORG 143, $0300
debugger 2
decimal 115-17
:DECINB 265, $DCC1
definition
in language creation 33-34
DEG (See also XDDEG, :SDEG) 105, 145, 211
DEGFLG 145, $00FB
deleting lines 28
DEND 81-82
DIM (See also XDIM, :SDIM) 16-17, 66, 127
and "('')" operator 70, 197
effects on tables 16-17
execution 106-7, 210
DIMENSION TOO SMALL error (See also ERRDIM) 96
direct statement 32, 49, 51-52, 74
DIRFLG 26, 30-32, 144, $00A6
Disk Device Dependent Note
Command 100
division (See FDIV, FRDIV, ')')
DOS (See also XDOS, :SDOS) 105, 109
DOSLOC 143, $000A
DPEEK 122
DPOKE 122
DRAWTO (See also XDRAWTO, :SDRAWTO) 92-93, 146, 233
dev ex machina 40-41, 46
DSPFLG 143, $02FE
DST 81-82
DVVT 81-82

E

ECHNG 45, 154, $A2BA
Editor (See Program Editor)
:EGTOK (See GETTOK)
ELADV 83, 235, $BADD
END (See also XEND, :SEND) 71-72, 225
End Of Expression (See EOE operator)
end-of-statement token (See also EOS) 76
ENDSTAR 139, 143, $008E
ENDVVT 143, $0088
English 37
ENTDTD 85-86, 110, $00B4
ENTER (See also XENTER, :SENTER) 23, 25, 123, 128, 140
device 71, 85-86
execution 85-86, 235
EOE operator 57-66
EOL character 58, 95-96, 98, 99-100, 143, 177
EOPUSH 65, 189, $AB15
EOS 169, $A6F8
EOS2 173, $A773
EPCHAR 143
equates
ICCOM value 146
ICSTA value 146
miscellaneous 143
Run Stack 147
variables 147
ERBRTN 230, $B920
ERGFDE 230, $B922
ERGFDEL 77, 79, 224, $B74A
ERLTL 230, $B924
ERNOFOR 78, 230, $B926
ERNOLN 75, 230, $B928
ERNTV 44, 152, $A201
ERON 230, $B93E
EROVFL 230, $B92A
ERRAOS 230, $B92C
ERRDIM 106, 230, $B92E
ERRDNO 230, $B918
ERRINP 96, 230, $B930
ERRLN 230, $B932
:ERRM1 231, $B961
:ERRM2 71, 231, $B974
ERRNSF 83, 230, $B916
ERRNUM 72, 101, 110, 144, $00B9
ERRROO 104, 230, $B934
ERROR 71, 88, 101, 106, 231, $B940
error handling 73-74
and DATA-READ 104
and DIM 106
and GOTO 75
and INPUT 96
and LOAD 83

Index
Index

and SETCOLOR 91
and SOUND 93
and TRAP 73, 231
execution 231
in I/O 101, 109, 146
in line processing 25-26, 28
in LISTing 88
in statement processing 29-30
in syntactical analysis 38-39
messages 230
missing FOR entry 78
missing GOSUB entry 79
missing line number 77
that stops program 73-74
ERRPTL 83, 230, $B91A
ERRSAV 144, $00C3
ERVAL 91, 93, 230, $B93A
ESRT (in ABML) 40-41, 44, 47, 162
:EVEN 249, $D8CE
EXECNL 32, 49, 75, 183, $A95F
EXECNS 183, $A962
Execute Expression (See also EXEXPR) 12, 55-70, 101, 109, 146, 189-90
Execution Control 49-54, 75, 77, 83, 183-85
Execution of GARBAGE error (See also XERR) 106
executor (See Program Executor) EXEOl 184, $A989
EXEXPR 64-65, 189, $AAE0
EXNXt 65, 189, $AAE3
EXOP 65, 69, 190, $AB20
EXPOP 65, 189, $AB0B
EXOT 65, 189, $AAEE
EXP (See also XPEXP) 180, 208
:EXP 162, $A60D
:EXP[X] 268, $DCC0
EXP10[X] 268, $DCCC
EXPAND 20-21, 106-7, 144, 181, $A881, Appendix B
EXPINT 65, 190, $AB2E
EXPTST 65, 189, $AAFA
EXSVOP 65, 144, $00AB
EXSVPR 144, $00AC
External Subroutine Call (See ESRT)
F
FADD 255, $DA66
FAIL 44-45, 153, $A26C
false (See XFALSE, :FALSE) :FALSE 224, $B788
FDB: Appendix A
FDIV 259, $DB28
files
LET-ENTER format (See FLIST)
SAVE-LOAD format 81-82
FIXRSTK 226-27, $BB82
FLIST 235, $BADA
floating point 126-27
add (See FADD, FRADD) ASCII to fp conversion 145, $00ED-$00F1
fp to ASCII conversion 250-52, $D8E6-$D9A9
fp to integer conversion 253-55, $D9D2-$DA5F
comparisons 196-97
divide (See FDIV, FRDIV) in ROM 14, 143, 246-72,
$D800-$DFF6
integer to fp conversion 252-53,
$D9AA-$D9CF
load/store 267
multiply (See FMUL, FRMUL)
routines 255-67
subtract (See FSUB, FRSUB)
zero page work area 143, 145,
$00D2-$00EC
FLOG 270, $DF01
FMOVER 183, Appendix B, $A947
FMUL 257 $DADB
:FNZERO 264, $DCA4
FOR (See also XFOR, :SFOR) 12
entry on Runtime Stack 18-19, 194, 133-34
execution 77-78, 220-21
tricks with 131
FPI 253, $D9D2
FPOR 143, 246, $D800
FR 145, $00D4
FR0 145, $00D5
FR1 145, $00E0
FR1M 145, $00E1
FR2 145, $00E6
FRA (i.e., FRA10, FRA20) 266, $D001
FRADD 196, $AD3B

288
FRCMP 196, $AD35
FRDIV 196, $AD4D
FRE
floating point memory location
145, $00DA
function (See also XPFRE) 180, 204
FRMUL 196, $AD47
FRSUB 196, $AD41
FS 172, $A751
FSUB 255, $DA60
Function Name Table 180
functions (See entry under individual
function name)

G
GD101 101, 239, $BC22
GDVCIO 100-101, 239, $BC1D
GET (See also XGET, :SGET) 97, 146,
239, Appendix B
GET1 239, Appendix B, SBC82
GETINT 192, $ABE9
GETADR 44, 152, $A215
:GETCHAR 260, $DB94
:GETDIG 265, $DCA9
GETINT 99, 192, $ABE0
GETLL 31, 53, 185, $A9DD
GETLN 151, $A19F
GETPRINT 192, $ABD5
GETSTMT 31, 52-53, 54, 75, 87, 103-4,
184, $A9A2
GETTOK 128, 190, $AB3E
:GETTOK 77, 79, 223, $B373
GETVAR 191, $AB89
GIOCMD 99, 241, $BDB0
GIODVC 99, 240, $BCC9
GIOPRM 241, $BD02
GLINE 234, $BAA9
GLINE 234, $BAA9
GNXTL 31, 51, 53, 185, $A9D0
GOSUB (See also XGOSUB, :SGOSUB) 12,
43, 79-80, 103, 127
entry on Runtime Stack 18-19,
133-34
execution 79, 221-22
in ABML 40, 42
in Operator Name Table 178
GOTO (See also XORO, :SGOTO) 75,
123, 128, 177, 221-22
GRAPHICS (See also XG, :SGR) 86, 91,
234
grammar 33-35, 37
:GR205, $B203
GRFBAS 143, $0270
GSTRAD 191, $AB9B
GVVTADR 193, $AC28

H
hexadecimal 2, 115-17
high level languages 3
high memory address 13
HIMEM 143, $02E5
HMADR 13, 143, $02E5

I
ICOM 146, $0342
ICOM 146, $0343
IF (See also XIF, :SIF) 76, 154, 224
IF 174, $A799
INBUFF 23, 25, 47, 88, 91, 98
INPUT (See also XINPUT, :SINPUT)
95-96, 143, 145, 213-14, Appendix B
INT (See also XPINT) 180, 206
interpreter 1, 3-5
INVAR 46
I/O 91-93, 95-102, 109, 234-43
I/O Call Routine 101-2, 241, $BDB0
IOC 9 (i.e., IOCB 0, ICB 1, etc.)
85-87, 91-92, 95, 99-101, 105, 110
close all (See CLSALL)
control block 146, $0340-$0350
ICOM value equates 146
ICOM value equates 146
IOCBORG 143, $0340
IOMOD 99, 102, 144, $00C0
IODVC 144, $00C1
I/O 101, 102, etc.) 83, 100, 101-2
IOTEST 86, 91, 93, 100, 101, 240, $BCCB
ISVAR 46, 241, $BD2F
ISVAR 46, 100, 241, $BD2D

J
joysticks (See STICK, :STRIG)
JS Appendix A

L
labels 34
language
creation of 33-39
problems with 1-2
high level 3
LBUFF 23, 25, 30, 145, $0580
LDD 240, $BADA
LDIOSTA 241, $BCEF
LELNUM 87, 144, $00AD
LEN (See also XPLEN) 180, 203
LET (See also Execute Expression, XLET, :SLET) 8, 11, 29, 42, 105
LIFO (last-in, first-out) stack 12, 43, 133
Line Buffer 23, 25-26
line number 27-28, 49-50, 52
line processing 25-28, 31-32, 95-96
LINE TOO LONG error (See also ERLTL) 25-26, 43, 48
LIST (See also XLIST, :SLIST) 15, 25, 123, 128, 144, 105
device 71, 242
entry in Runtime Stack 19
execution 86-87, 216-17, 222
subroutines 88-90
LISTDTD 86-87, 110, 123, 128, 139-40
as block (See LSBLK) execution 83-84, 236
file format 81-82
LOADFLG 109-110, 144, $00CA
LOCAL: Appendix A
LOCATE (See also XLOCATE, :SLOCATE) 92, 240, Appendix B
LOG (See also XLOG) 180, 208, 270, $DECD
LOG10 (See also XPL10) 208
LOG10(X) 270, $DED1
LOMEM 23, 115
L1 176, $A7C0
low memory address 13
LPRINT (See also XLPINT, :SLPRINT) 98-99, 216
:LPRTOKEN 71, 88-89, 90, 218, $B535
:LPTWB 218, $B54F
LSLBLK 237, $B88B
:LSCAN 89-90, 217-18, $B50C
LSRA: Appendix A
:LSTM 88-89, 219-20, $B590
L2 176, $A7C4
MEMTOP 20-21, 143, $0090
MEOLFLG 143, $0092
meta-language 33
MOD (See modulo) modulo 120-22
multiplication (See FMUL, FRMUL, **) multipurpose buffer 13, 65, 82, 110
:MV6RS 228, $B88F
MVFR1E 267, $DD34
MVFR12 266, $DD28
MVNLG 183, Appendix B

NEXXT (See also SNEXT) 18, 131
execution 78-79, 222-23
in Pre-compiler 43-45, 151, $A1E2
NEW (See also XNEW, :SNEW) 109-110, 123, 128
NFP 164, $A672
NFSP 176, $A7CE
NFUN 164, $A65F
NFUSR 164, $A669
NIBSHO 261, $DBEB
NMAT 164, $A651
NMAT2 164, $A659
non-terminal 35, 40
NOP 163, $A62E
NORM 262, $DC00
NOT (See also XPNOT) 178, Appendix B
NOTE (See also XNOTE, :SNOTE) 100, 239
NSMAT 173, $A777
NSML 174, $A78C
NSML2 174, $A790
NSVAR 169, $A708
NSVRL 169, $A710
NSV2 170, $A714
NULL (in ABML) 162
numeric constants 89, 131
numeric variable 7-8, 95
NV 162, $A622
NVAR 163, $A64C
NXSC 44, 153, $A2A1
NXSTID 50, 54, 72, 144, $00A7

M
machine language 1-2
macros Appendix A
mantissa 127
MAXCIX 26, 29, 144, $009F
MDESUP 265, $DCE0
MEMFULL 230, $B93C
memo pad 105
memory
management routines 20-23
organization 13-14
pointer addresses 13, 83
MEMTOP 20-21, 143, $0090
MEOLFLG 143, $0092
meta-language 33
MOD (See modulo) modulo 120-22
multiplication (See FMUL, FRMUL, **) multipurpose buffer 13, 65, 82, 110
:MV6RS 228, $B88F
MVFR1E 267, $DD34
MVFR12 266, $DD28
MVNLG 183, Appendix B

N
NEXT (See also SNEXT) 18, 131
execution 78-79, 222-23
in Pre-compiler 43-45, 151, $A1E2
NEW (See also XNEW, :SNEW) 109-110, 123, 128
NFP 164, $A672
NFSP 176, $A7CE
NFUN 164, $A65F
NFUSR 164, $A669
NIBSHO 261, $DBEB
NMAT 164, $A651
NMAT2 164, $A659
non-terminal 35, 40
NOP 163, $A62E
NORM 262, $DC00
NOT (See also XPNOT) 178, Appendix B
NOTE (See also XNOTE, :SNOTE) 100, 239
NSMAT 173, $A777
NSML 174, $A78C
NSML2 174, $A790
NSVAR 169, $A708
NSVRL 169, $A710
NSV2 170, $A714
NULL (in ABML) 162
numeric constants 89, 131
numeric variable 7-8, 95
NV 162, $A622
NVAR 163, $A64C
NXSC 44, 153, $A2A1
NXSTID 50, 54, 72, 144, $00A7
Index

REM (See also XREM, :SREM) 106, 131, 140, 154
RESTORE (See also XRESTORE, :SRESTORE) 103-4, 211
RETURN (See also XRTN, :SRET) 12, 18, 79-80, 223-24
Return (in ABML) 41
:REXPAN 228, $B878
Richard's Rule Appendix B
RISASN 96
RND (See also XPRND) 37, 69, 180, 206
RNDLOC 143, $D20A
RNTV44
ROLA: Appendix A
ROM 143, $A000
ROM tables 9-10, 135-36
RORA: Appendix A
RSHFOE 263, $DC62
RSHFTO 263, $DC3A
RSHFTl 263, $DC3E
RSTPTR 229, $B8AF
RSTSEOL 100, 242, $BD99
RTN (in ABML) 41, 44-45, 162
RETURN 41, 44-45, 162
RSTVAR 96, 193, $AC16
RTS 45, 51, 96, 106
RUN (See also XRUN, :SRUN, RUNINIT)
as direct statement 49, 52
execution 71-73, 224
initialization 103, 105
with implied LOAD (See also FRUN) 235
RUNINIT 230, $B8F8
RUNSTK 19, 143, $008E
Runtime Stack 10, 14
and FOR, NEXT, GOSUB,
RETURN 76-80
entry format 18-19
listing 133-34
pointer to 19
scalar 126
SCANT 89-90, 97-98, 144, $00AF
:SCALLD 167, $A6BE
:SCLOSE 170, $A721
:SCLR 167, $A6BE
:SCOLOR 167, $A6BD
:SCOM 173, $A760
:SCONT 167, $A6BE
:SCADR 90
screen editor 25
SCRI 92, 143, $0055
SCRY 92, 143, $0054
:SCSAVE 167, $A6BE
SCVECT 272, $BFF9
:SDATA 176, $A7CB
:SG16 167, $A6BE
:SDIM 173, $A760
:SDOS 167, $A6BE
:SDRAWTO 172, $A75D
SEARCH 29, 47, 135, 158, $A462
:SEND 167, $A6BE
:SENDER 170, $A724
:SETCODE 27-29, 154, $A2C8
:SETCOLOR (See also XSETCOLOR, :SSETCOLOR) 91-92, 232
SETDZ 242, $BD72
SETLINE 54, 226, $B818
SETLNI 50, 54, 226, $B81B
SETSEOL 99, 242, $BD79
SFNP 177, $A76D
:SFOR 167, $A6D2
:SFUN 165, $A678
:SGOSUB 167, $A6BD
:SGOTO 167, $A6BD
:SGR 167, $A6BD
:SIKIO 101, 240, $BCB9
:SID 174, $A794
SIN (See also XPSIN, SIN[X]) 105, 180, 207, 243
SIN[X] 243, $BDA7
:INPUT 169, Appendix B, $A6F4
SKBLANK 29, 261, $DBA1
SKCTL 143, $D20F
SKPBLANK 260-61, $DBA1
:SLET 167, $A6C0
SLIS 171, $A73C
:S LIST 171, $A733
:SLOAD 170, $A724
:SLocate 168, $A6E2
:SLPRINT 169, $A700
SMAT 165, $A694
SMAT2 166, $A69C
:SNEW 167, $A6BE

SADR 66-68
SAP (Simple Arithmetic Process) 33-39
SACUR 144, $00BE
:SAVEX 228, $888A
SAVE (See also XSAVE, :SSAVE) 81-83, 85-86, 139
as block (See LSBLK)
execution 82-83, 237
file format 81-82
SAVE 'C:' 84
SAVEOP 57, 59-64
:SAVE TOP 228, $B881
:SBYE 167, $A6BE

S
SADR 66-68
SAP (Simple Arithmetic Process) 33-39
SACUR 144, $00BE
:SAVEX 228, $888A
SAVE (See also XSAVE, :SSAVE) 81-83, 85-86, 139
as block (See LSBLK)
execution 82-83, 237
file format 81-82
SAVE 'C:' 84
SAVEOP 57, 59-64
:SAVE TOP 228, $B881
:SBYE 167, $A6BE •
Index

:SNEXT 168, $A6EA
:SNOTE 172, $A74A
SNTAB (See also Statement Name Table) 115, 135-36, 159, $A4AF
SNX286, 148, $A053
SOP 166, $A6A2
SOPEN 170, $A71A
SOUND (See also XSOUND, :SSOUND) 93, 232
sound registers (See also SREGn, :SKCTL) 93, 143
:SPLOT 172, $A75D
:SPONENT 172, $A74A
:SPRESH 172, $A75D
:SPRINT 169, $A6FC
:SPUT 166, $A6BA
speed comparisons 3-5
SQR (See also XPSQR, SQR[X]) 180, 208, 245
SQR[X] 245, $BEE3
:SRAD 167, $A6BE
SRCADR 29, 43, 48, 144, $0095
SRCOUNT 45-46, 154, $A2E6
:SUREAD 169, Appendix B, $A6F5
SREGn (i.e., SREG1, SREG2, etc.) 143, $D208, $D201-2
:SREMS 167, $A7C8
:SRESET 168, $A6EF
:SREX 167, $A6BE
:SRUN 170, $A727
:SSAVE 170, $A724
:SSETCOLOR 172, $A75B
:SOUND 172, $A759
:SSTATUS 171, $A741
:SSTOP 167, $A6BE
ST (See Statement Table)
stack (See also Argument Stack, Operator Stack, Runtime Stack, CPU stack) 2, 12
STACK OVERFLOW error (See also ERRAOS) 66
STARP 18, 139, 143, $008C
Start Of Expression (See SOE Operator)
STAT 171, $A744
statement execution 50-51
processing 28-31
Statement Execution Table (See also STETAB) 9, 185-87
Statement Name Table (See also SNTAB) 9, 40, 135-36, 159-61
Statement Name Token 8, 12, 106
Statement Syntax Table 10-11, 33, 40
Statement Table 10-11, 14, 49-50, 52
entry format 17, 131
in LIST 87-88
in NEW 110
in SAVE and LOAD 81-82
listing in token form 129-31
processing 31
STATUS (See also XSTATUS, :SSTATUS) 100, 146, 239
:STCHAR 264, $DC9F
STCOMP 165, $A67E
STENUM 29, 48, 144, $00AF
STEP execution 77-78
in Operator Name Table 178
in Runtime Stack 19
STETAB (See also Statement Execution Table) 185, $AA00
STICK (See also XPSTICK) 180, 204
STINDEX 65, 87-88, 144, $00A8
STKVL 43, 144, $00A9
STMLBD 29-30, 44, $00A7
STMDB 17, 131, 143, $0088
STMCUR 20, 31-32, 49-54, 64, 72, 88, 143, $008A
STNUM 29, 48, $00A8
:STNUM 264, $DC9D
STOP (See also XSTOP, :SSTOP) 50, 71-72, 124, 225
STOPLN 71-72, 110, $00BA
STR (See also XPSTR)
function 205
routine 165, $A682
STR$180
:STRAP 167, $A6BD
STRCMP 202, $A8F1
STRIG (See also XPSTRIG) 180, 204
String/Array Table 10, 106-7, 127
pointers into 15-16, 143
entry format 18
SAVEing 139
use of, in Execute Expression 66-67
string assign operator 200-202
bug Appendix B
comparisons (See STRCMP)
constants (literals) 89, 131
variables 15-18, 66-67, 70, 96, 107
subscripts (See arrays, ‘(‘ and ‘)’)
subtraction (See FSUB, FRSUB, ‘-‘)
SVAR 165, $A68F
SVCOLOR 92, 143, $02FB
SVDISP 77, 79, 144, $00B2

293
Index

SVVNTP 26, 144, $00AD
SVVVTE 27, 144, $00B1
:SXIO 170, $A718
symbols 1, 4
in language creation 33-39
SYN: Appendix A
SYNENT 42, 151, $A1C3
SYNTAX 74, 148, $A060
syntax 1, 10-11, 23, 29-30
analysis of 37-39
bugs with Appendix B
creation of 35-39
traces with Appendix B
memory organization 148-53
tables 40-42, 162-77
syntaxer (See Pre-compiler)
Syntax Stack 23, 43

T

tables 9-10
Function Name Table 180
Operator Execution Table 9,
187-89
Operator Name Table 9, 46, 89,
135-36, 177-80
Operator Precedence Table 9-10,
56, 137-38, 193-94, Appendix B
RAM tables 11, 81-82, 110, 143
ROM tables 9-10
Runtime Stack 10, 14, 18-19,
76-80, 133-34
Statement Execution Table 9,
185-87
Statement Name Table 9, 40,
135-36, 159-61
Statement Syntax Table 10-11
Statement Table 10, 14, 17, 31,
49-50, 52, 81-82, 110, 129-31
String/Array Table 10, 13, 18,
106-7, 127
syntax tables 40-42, 162-77
Variable Name Table 10, 13, 15,
26, 46, 81-82, 110, 123, 135-36
Variable Value Table 10, 13, 15-16,
27, 46, 66-67, 78, 81-82, 106-7,
125-28
TENDST 51, 52, 185, $A9E2
terminal symbol 34-36
TERMTST 44-45, 154, $A2A9
TEXP 172, $A755
THEN (See also XIF, :SIF) 58, 76, 178
TNCON 47, 157, $A400
TNVAR 46, 155, $A32A
TO 131, 178
tokens 5-8, 15, 17, 88-89, 135
TOPRSTK 143, $0090
transcendental functions 207-9
translators 1-3
TRAP (See also XTRAP, :STRAP) 73, 76,
225, 231
TRAPLN 73, 76, 110, 144, $00BC
ttrue (See XTRUE)
TSTALPH 157, $A3F3
:TSTCHAR 261, $DBBB
TSTEND 230, $B910
TSTNUM 261, $DBAF
TVAR 155, $A32E
TVAR 155, $A330

U

UNARY 162, $A618
unary + and - 179, Appendix B
USR (See also XPUSR) 180, 206

V

VAL (See also XPVAL) 180, 204
Variable Name Table 10, 13, 26, 46
e ntry format 15
in NEW 110
in SAVE and LOAD 81-82
listing 123-24, 135-36
variables (See also numeric v, string
v, array v) 8, 95
finding and listing 139-40
listing 123-28
tokens 8, 88-89
Variable Value Table 10, 13, 27, 46,
66-67, 78, 106-7
e ntry format 15-17
in SAVE and LOAD 81-82
listing 125-28
VEXP (in ABML) 41, 44-45, 162
VNT (See Variable Name Table)
VNTD 20, 123, 143, $008D
VNTP 15, 123, 139, 143, $008E
VNUM 66-68
VVT (See Variable Value Table)
VVTP 17, 143, $0086

W

WARMFLG 109-110, 143, $0008
WARMSTART 109-110, 148, $A04D
WORD 122
WVVPT 144, $009D
Index

XPPLUS 194, $ACB4
XPUSR 206, $B0BA
XPUT 99, 239, $BC72
XPVAL 204, $B000
XPxxx (i.e., operator and function execution routines) 69, 194-97, 203-9
XRAD 105, 211, $B266
XRD3 96, 212, $B2D0
XREAD 103-4, 211, $B283
XREM 106, 185, $A9E7
XREST 104, 211, $B26B
XRTN 79, 87, 104, 223, $B719
XRUN 51, 71-73, 224, $B74D
XSAASN 200, $AEA3
XSAVE 82-83, 237, $BB50
XSETCOLOR 91-92, 232, $B9B7
XSOUND 93, 232, $B9DD
XSPV 200, $AE96
XSTATUS 100, 239, $BC28
XSTOP 50, 71-72, 96, 225, $B793
XTRAP 76, 225, $B7E1
XTRUE 195, $AD05
XXIO 99, 238, $BBE5

Y

y index 69-70

Z

Z = |X-C|/(X+C) (See also XFORM) 269-70
zero default with DIM 127
zero page
  floating point work area 143
  pointers 20, 110, 143-44
  RAM locations 144
ZFP 143, $00D2
ZICB 143, $0020
ZPADEC 203, $AFBC
ZPG1 143, $0080
ZVAR 229, $B8C0
If you've enjoyed the articles in this book, you'll find the same style and quality in every monthly issue of COMPUTE! Magazine. Use this form to order your subscription to COMPUTE!

For Fastest Service, Call Our Toll-Free US Order Line
800-334-0868
In NC call 919-275-9809

COMPUTE!
P.O. Box 5406
Greensboro, NC 27403

My Computer Is:
□ PET □ Apple □ Atari □ VIC □ Other ______ □ Don't yet have one...

□ $20.00 One Year US Subscription
□ $36.00 Two Year US Subscription
□ $54.00 Three Year US Subscription

Subscription rates outside the US:
□ $25.00 Canada
□ $38.00 Europe, Australia, New Zealand/Air Delivery
□ $48.00 Middle East, North Africa, Central America/Air Mail
□ $68.00 Elsewhere/Air Mail
□ $25.00 International Surface Mail (lengthy, unreliable delivery)

Name

Address

City

State

Zip

Country

Payment must be in US Funds drawn on a US Bank; International Money Order, or charge card.

□ Payment Enclosed
□ VISA
□ MasterCard
□ American Express

Acc t. No. ________________________
Expires ______ / ______

15-9
Ask your retailer for these COMPUTE! Books. If he or she has sold out, order directly from COMPUTE!

For Fastest Service
Call Our TOLL FREE US Order Line
800-334-0868
In NC call 919-275-9809

<table>
<thead>
<tr>
<th>Quantity</th>
<th>Title</th>
<th>Price</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>The Beginner's Guide to Buying A Personal Computer</td>
<td>$3.95**</td>
<td></td>
</tr>
<tr>
<td></td>
<td>COMPUTE!'s First Book of Atari</td>
<td>$12.95*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Inside Atari DOS</td>
<td>$19.95*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>COMPUTE!'s First Book of PET/CBM</td>
<td>$12.95*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Programming the PET/CBM</td>
<td>$24.95***</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Every Kid's First Book of Robots and Computers</td>
<td>$4.95**</td>
<td></td>
</tr>
<tr>
<td></td>
<td>COMPUTE!'s Second Book of Atari</td>
<td>$12.95*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>COMPUTE!'s First Book of VIC</td>
<td>$12.95*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>COMPUTE!'s First Book of Atari Graphics</td>
<td>$12.95*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mapping the Atari</td>
<td>$14.95*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Home Energy Applications On Your Personal Computer</td>
<td>$14.95*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Machine Language for Beginners</td>
<td>$12.95*</td>
<td></td>
</tr>
</tbody>
</table>

* Add $2 shipping and handling. Outside US add $5 air mail; $2 surface mail.
** Add $1 shipping and handling. Outside US add $5 air mail; $2 surface mail.
*** Add $3 shipping and handling. Outside US add $10 air mail; $3 surface mail.

Please add shipping and handling for each book ordered.

Total enclosed or to be charged.

All orders must be prepaid (money order, check, or charge). All payments must be in US funds. NC residents add 4% sales tax.

Payment enclosed Please charge my: ☐ VISA ☐ MasterCard ☐ American Express Acc't. No. Expires /

Name
Address
City State Zip
Country

Allow 4-5 weeks for delivery.
The Atari® BASIC Sourcebook

Everything You Always Wanted to Know about the Making of a Computer Language

- When you type in BASIC programs and run them, what is really going on inside the computer?
- How does the computer know how to handle a FOR-NEXT loop and where it should go when it meets a RETURN?
- Where do ERROR messages come from?
- How does the computer decide which mathematical operation to perform first?
- Why do some processes take so long, when others are almost instantaneous?
- What sometimes causes the computer to lock up when you delete lines from your program?
- How does the computer know what to do when it sees words and symbols like GOTO, INT, CHR$, *, +, and >?
- How can your machine language programs take advantage of some of the sophisticated routines in Atari BASIC?

The creators of Atari BASIC have now revealed their own work. Even if you aren't a machine language programmer, you'll find this book a fascinating exploration of a computer language. Now you can understand exactly why your programs work as they do. And if you are a machine language user, the source listing will let you see exactly where to enter Atari BASIC to use the powerful routines built into the language.


$12.95