The Second Book Of Machine Language

Personal Computer Machine Language Programming for the Commodore 64, VIC-20, Atari, Apple, and PET/CBM Computers

By Richard Mansfield

A COMPUTE! Books Publication

The Second Book of Machine Language

By Richard Mansfield

COMPUTE!™ Publications, Inc. abc
One of the ABC Publishing Companies
Greensboro, North Carolina

Printed in the United States of America

ISBN 0-942386-53-1

10 9 8 7 6 5 4 3 2 1

COMPUTE! Publications, Inc., Post Office Box 5406, Greensboro, NC 27403, (919) 275-9809, is one of the ABC Publishing Companies, and is not associated with any manufacturer of personal computers. PET, CBM, VIC-20, and Commodore 64 are all trademarks of Commodore Electronics Limited and/or Commodore Business Machines, Inc. Apple is a trademark of Apple Computer Company. Atari is a trademark of Atari, Inc.

Contents

Preface

This book shows how to put together a large machine language program. All of the fundamentals were covered in my first book, *Machine Language for Beginners*. What remains is to put the rules to use by constructing a working program, to take the theory into the field and show how machine language is done.

Showing how to construct an assembler—written entirely in machine language—would serve two useful purposes. It would illustrate advanced programming technique and also provide the reader with a powerful assembler to use in other ML programming.

This book, then, offers the reader both a detailed description of a sophisticated machine language program (the LADS assembler) and an efficient tool, a complete language with which to write other machine language programs. Every line in the LADS assembler program is described. All the subroutines are picked apart and explained. Each major routine is examined in depth.

LADS, the Label Assembler Development System, is a fast, feature-laden assembler—it compares favorably with the best assemblers available commercially. And not the least of its virtues is the fact that few programs you will ever use will be as thoroughly documented and therefore as accessible to your understanding, modification, and customization.

LADS is a learning device too. By exploring the assembler, you will learn how to go about writing your own large machine language (ML) programs. You will see how a data base is created and maintained, how to communicate with peripherals, and how to accomplish many other ML tasks. Also, because you can study the creation of a computer language, the LADS assembler, you will gain an in-depth knowledge of the intimate details of direct communication with your computer.

Most programming involves a tradeoff between three possible objectives: speed, brevity, or clarity. You can program with the goal of creating the fastest running program possible. Or you can try to write a program which uses up as little memory as possible. Or you can try to make the program as understandable as possible, maximizing the readability of the program listing with REMarks.

LADS emphasizes clarity so that its source code will serve as a learning tool and as the focus of this book. It's designed so that important events in the program can be easily explained and understood. Virtually every ML instruction, every tiny step, is commented within the source code listings following each chapter.

This doesn't mean that LADS is flabby or slow. Assembling roughly 1000 bytes a minute and taking up 5K in memory, LADS is considerably faster and more compact than most commercial assemblers. That's because, in ML, you can have the best of both worlds: You can comment as heavily as you want, but the assembler will strip off the comments when it creates the object code. In this way, clarity does not sacrifice memory or speed.

The frequent comments contribute considerably to the educational value of this assembler. Exploring LADS is a way to learn how to achieve many common programming goals and how to construct a large, significant program entirely in ML. An additional advantage of this comprehensibility is that you'll be able to modify LADS to suit yourself: Add your own pseudo-ops, define defaults, format output. All this is referred to as a language's *extensibility*. We'll get to this in a minute.

What BASIC is to BASIC programming, an assembler is to ML programming. LADS is a complete language. You write programs (source code) which LADS translates into the finished, executable ML (object code). Unlike less advanced assemblers, however, symbolic assemblers such as LADS can be as easy to use as higher level languages like BASIC. The source code is very simple to modify. Variables and subroutines have names. The program can be internally commented with REM-like explanations. Strings are automatic via the .BYTE command. There are a variety of other built-in features, the pseudo-ops, which make it easy to save object programs, control the screen and printer listings, choose hex or decimal disassembly, and service other common programming needs.

Perhaps the best feature of LADS, though, is its *extensibility*. Because you have the entire source code along with detailed explanations of all the routines, you can customize

LADS to suit yourself. Add as many pseudo-ops as you want. Redesign your ML programming language anytime and for any reason. Using an extensible programming language gives you control not only over the programs you design, but also over the way that they are created. You can adjust your tools to fit your own work style.

Do you often need to subtract hex numbers during assembly? It's easy to stick in a – command. Would you rather that LADS read source programs from RAM memory instead of disk files? (This makes it possible to assemble using a tape drive. It can also be a bit faster.) In Chapter 11 we'll go through the steps necessary to make this and other modifications. You'll be surprised at how easy it is.

Finally, studying the language (the LADS assembler) which produces machine language will significantly deepen your understanding of ML programming.

I would like to thank Charles Brannon for his translation and work with the Atari version of LADS, Kevin Martin for his translation and work with the Apple version, and Todd Heimarck for his many helpful discoveries about the assembler.

Chapter 1

How to Use This Book

How to Use This Book

The dual nature of this book—it's both a text and a program—offers you a choice. You can follow the ideas: reading through the chapters, studying the program listings, and deepening your understanding of machine language programming.

Alternatively, you can type in the LADS assembler and experiment with it: learning its features, trying out modifications, and using it to write your own machine language programs. Appendix A describes how to use the assembler and Appendix B provides instructions on typing it in. If you choose this second approach, the rest of the book can serve as a reference and a map for modifying the assembler. The tutorials can also help to clarify the structure and purpose of the various subroutines and subprograms.

LADS is nearly 5K long, and for those who prefer not to type it in, it can be purchased on a disk by calling COMPUTE! Publications toll free at 1-800-334-0868. Be sure to state whether you want the Commodore, Atari, or Apple disk. The disk contains both the LADS source and object code (these terms are defined below). To create customized versions of the assembler, you will need the source code. It, too, can be typed in (it is printed in sections at the end of Chapters 2–9). If you don't type in any of the comments, it is roughly 10K long. The Commodore disk contains the various PET/CBM (Upgrade and 4.0 BASIC), VIC, and Commodore 64 versions.

Definitions

There are several concepts and terms which will be important to your understanding of the rest of the book.

ML programming, and programming in general for that matter, is a new discipline, a new art. There are few rules yet and few definitions. Words take on new meanings and are sometimes used haphazardly. For example, the word *monitor* means two entirely different things in current computerese: (1) a debugging program for machine language work or (2) a special TV designed to receive video signals from a direct video source like a computer.

Since there is no established vocabulary, some programming ideas are described by an imprecise cluster of words. When applied to machine language programming, the terms *pointer, variable, register, vector, flag,* and *constant* can all refer to the same thing. There are shades of difference developing which distinguish between these words, but as yet, nothing has really solidified. All these terms refer, in ML parlance, to a byte or two which the programmer sets aside in the source code. In BASIC, all these terms would be covered by the word *variable*.

Loose Lingo

Purists will argue that each of these words has a distinct, definable meaning. But then purists will always argue. The fact is that computing is still a young discipline and its lingo is still loose.

Some professors of BASIC like to distinguish between *variables* and *constants*, the latter meaning unchanging definitions like SCREEN = 1024. The address of the start of screen RAM is not going to vary; it's a constant.

In BASIC, something like SCORE = 10 would be a variable. The score might change and become 20 or whatever. At any rate, the word SCORE will probably vary during the execution of the program. In ML, such a variable would be set up as a two-byte reserved space within the source code:

100 SCORE .BYTE 0 0

Then, anytime you ADC SCORE or ADC SCORE+1, you will add to the SCORE. That's a variable. The word *pointer* refers to those two-byte spaces in zero page which are used by Indirect Y addressing—like LDA (155),Y—and which serve to point to some other address in memory.

Register usually means the X or Y or Accumulator bytes within the 6502 chip itself. As generally used, the word *register* refers to something hard wired within the computer: a circuit which, like memory, can hold information. It can also refer to a programmer-defined, heavily used, single-byte variable within an ML program:

100 TEMP .BYTE 0

A *vector* is very much like a *pointer*. It stores a two-byte address but can also include the JMP instruction, forming a three-byte unit. If you have a series of *vectors*, it would be called a "jump table," and the Kernal in Commodore computers is such a table:

```
FFD2 JMP $F252
FFD5 JMP $A522
FFD8 JMP $B095
```

Thus, if you JSR $FFD2, you will bounce off the JMP into $F252, which is a subroutine ending in RTS. The RTS will send you back to your own ML code where you JSRed to the JMP table. That's because JMP leaves no return address, but JSR does.

A *flag* is a very limited kind of variable: It generally has only two states, on or off. In LADS, PRINTFLAG will send object code (defined below) to the printer if the flag holds any number other than zero. If the PRINTFLAG is down, or off, and holds a zero, nothing is sent to the printer. The word *flag* comes from the Status Register (a part of the internals of the 6502 chip). The Status Register is one byte, but most of the bits in that byte represent different conditions (the current action in an ML program resulted in a negative, a zero, a carry, an interrupt, decimal mode, or an overflow). The bits in the Status Register byte are, themselves, individual flags. ML programmers, however, usually devote an entire byte to the flags they use in their own programs. Whole bytes are easier to test.

Source code is what you type into the computer as ML instructions and their arguments:

```
100 *= 864
110 LDA #$0F ; THIS WILL PUT A 15 ($0F) INTO THE
             ACCUMULATOR
120 INY      ; THIS RAISES THE Y REGISTER
```

After you type this in, you *assemble* it by turning control over to the LADS assembler after naming this as the source code. The result of the assembly is the *object code.* If you have the .S pseudo-op on, causing the object code to print to the screen, you will see:

```
100 0360 A9 0F      LDA #$0F   ; THIS WILL PUT A 15 ($0F)
                               INTO THE ACCUMULATOR
120 0362 C8         INY        ; THIS RAISES THE Y
                               REGISTER
```

Properly speaking, the object code is the numbers which, taken together, form a runnable ML program. These numbers can be executed by the computer since they are a program. In the example above, the object code is A9 0F C8. That's the computer-understandable version of LDA #$0F: INY. It's generated by the assembler. An assembler translates source code into object code.

A complex assembler like LADS allows the programmer to use labels instead of numbers. This has several advantages. But it does require that the assembler pass through the source code *twice*. (When an assembler goes through source code, it is called a *pass*.) The first time through, the assembler just gathers all the label names and assigns a numeric value to each label. Then, the second time through the source code, the assembler can fill in all the labels with the appropriate numbers. It doesn't always know, the first time through, what *every* label means. Here's why:

```
100 LDA 4222
110 BEQ NOSCORE
120 JMP SOMESCORE
130 NOSCORE INX:JMP CONTINUE
140 SOMESCORE INY
150 CONTINUE LDA 4223
```

As you can see, the first time the assembler goes through this source code, it will come upon several labels that it doesn't yet recognize. When the assembler is making its first pass, the labels NOSCORE, SOMESCORE, and CONTINUE have no meaning. They haven't yet been defined. They are *address-type* labels. That is, they stand for a *location* within the ML program to which JMPs or branches are directed. Sometimes those jumps and branches will be *forward* in the code, not yet encountered.

The assembler is keeping track of all the addresses as it works its way through the source code. But labels cannot be defined (given their numeric value) until they appear. So on the first pass through the source code, the assembler cannot fill in values for things like NOSCORE in line 110. It will do this the second time through the source code, on the *second pass*. The first pass has a simple purpose: The assembler must build an array of label names and their associated numeric values. Then, on the second pass, the assembler can look up each label in the array and replace label names (when they're being used as arguments like LDA NAME) with their numeric value. This transforms the words in the source code into numbers in the object code and we have a runnable ML program. Throughout this book, we'll frequently have occasion to mention pass 1 or pass 2.

The Two Kinds of Labels

There are two kinds of labels in ML source code: *equate* and *address* labels. Equate labels are essentially indistinguishable from the way that variables are defined in BASIC:

100 INCOME = 15000

This line could appear, unaltered, in LADS or in a BASIC program. (Remember this rule about labels: Define your equate labels at the start of the source code. The LADS source code shows how this is done. The first part of LADS is called Defs and it contains all the equate definitions. This is not only convenient and good programming practice; it also helps the assembler keep things straight.)

The other kind of label is not found in BASIC. It's as if you can give a name to a line. In BASIC, when you need to branch to a subroutine, you must:

```
10 GOSUB 500
.
.
500 (the subroutine sits here)
```

that is, you must refer to a line number. But in LADS, you give subroutines names:

```
10 JSR RAISEIT; GOSUB TO THE RAISE-THE-Y-REGISTER-
SUBROUTINE
.
.
500 RAISEIT INY; THE SUBROUTINE WHICH RAISES Y
510 RTS
```

This type of label, which refers to an address within the ML program (and is generally the target of JSR, JMP, or a branch instruction), is called an *address-type* label, or sometimes a *PC-type* label. (PC is short for Program Counter, the variable within the 6502 chip which keeps track of where we are during execution of an ML program. In LADS, we refer to the variable SA as the Program Counter—SA keeps track, for LADS, of where it is during the act of assembling a program.)

Subprogram is a useful word. LADS source code is written like a BASIC program, with line numbers and multiple-statement lines, and it's written in a BASIC environment. The source code is saved and loaded as if it were a BASIC program. But if you are writing a large ML program, you might write several of these source code "programs," saving them to disk separately, but linking them with the .FILE and .END pseudo-ops into one big chain of source programs. This chain will be assembled by LADS into a single, large, runnable ML object program.

Each of the source programs, each link in this chain, is called a *subprogram*. In the source code which makes up LADS there are 13 such subprograms—from Defs to Tables—comprising the whole of LADS when assembled together. This book is largely a description of these subprograms, and some chapters are devoted to the explication of a single subprogram. To distinguish subprograms from subroutines and label names, the subprogram names (like Tables) have only their first letter capitalized. Subroutines and labels are all-caps (like PRINTFLAG).

The word *integer* means a number with no fraction attached. In the number 10.557, the integer is the 10 since integers have no decimal point. They are whole numbers. ML programs rarely work with anything other than integers. In fact, the integers are usually between 0 and 65535 because that's a convenient range within which the 6502 chip can operate—two bytes can represent this range of numbers. Of course, decimal fractions are not allowed. But virtually anything can be accomplished with this limitation. And if you need to work with big or fractional numbers, there are ways.

In any case, when we refer to *integer* in this book, we mean a number that LADS can manipulate, in a form that LADS can understand, a number which is a *number* and not, for example, a graphics code. For example, when you write LDA $15 as a part of your source code, the computer holds the number 15 in ASCII code form. In this printable form, 15 is held in the computer as the numbers $31 $35 which, when printed on the screen, provide the *characters* 1 and 5 (but not the true number 15). For the assembler to work with this 15 as the number 15, it must be transformed into a two-byte *integer*, an actual number. When translated, and put into two bytes, the characters 1 5 become: $0F 00. We'll see what this means, and how the translation is accomplished, in Chapter 5 where we examine the subprogram Valdec. It's Valdec's job to turn ASCII characters into true numbers.

The Seventh Bit (Really the Eighth)

For most of human history, we had to get along without the 0. It was a great leap forward for mankind when calculations could include the concept of nothing, zero. But now there's another mental leap to be made, a private adjustment to the way that computers use zero: They often start counting with a zero, something humans never do.

Imagine you are driving along and you've been told that your friend's new house is the third house in the next block. You don't say "house zero, house one, house two, house three." It makes no sense (to us) to say "house zero." We always count up from 1.

But the computer often starts counting from zero. In BASIC, when you DIM (15) to dimension an array, it's easy to overlook the fact that you've really DIMed *16* items—the computer has created a *zeroth* item in this array.

It's sometimes important to be aware of this quirk. A number of programming errors result from forgetting that unnatural (or at least, nonhuman) zeroth item.

This situation has resulted in an unfortunate way of counting bits within bytes. It's unfortunate in two ways: Each bit is off by 1 (to our way of thinking) because there is a zeroth bit. And, to make things even tougher on us, the bits are counted *from right to left*. Quite a perversity, given that we read from left to right. Here's a diagram of the Status Register in the 6502 chip, each bit representing a flag:

7 6 5 4 3 2 1 0 (bit number within the Status Register byte)
N V - B D I Z C (flag name)

As a brief aside, let's quickly review the meanings of these flags. The flag names in the Status Register reflect various possible conditions following an ML event. For example, the LDA command always affects the N and Z flags. If you LDA #0, the Z flag will go up, showing that a zero resulted (but the N flag will go, or stay, down since the seventh bit isn't set by a zero). Here's what the individual flags mean: N (negative result), V (result overflowed), - (unused), B (BRK instruction used), D (decimal mode), I (interrupt disable), Z (result zero), C (carry occurred).

But in addition to the meanings of these flags in the Status Register, notice how bytes are divided into bits: count right to left, and start counting from the zeroth bit.

This is relevant to our discussion of LADS when we refer to bit 7. This bit has a special importance because it can signify several things in ML.

If you are using signed arithmetic (where numbers can be positive or negative), bit 7 tells you the sign of the number you're dealing with. In many character codes, a set (up) seventh bit will show that a character is shifted (that it's F instead of f). In the Atari, it means that the character is in inverse video. But a set seventh bit often signifies something.

One common trick is to use bit 7 to act as a delimiter, showing when one data item has ended and another begins. Since the entire alphabet can easily fit into numbers which don't require the seventh bit up (any number below 128 leaves the seventh bit down), you can set up a data table by "shifting" the first character of each data item to show where it starts. The data can later be restored to normal by "lowering" the shifted character. Such a table would look like this:

FirstwordSecondwordAnotherwordYetanother.

BASIC stores a table of all its keywords in a similar fashion, except that it shifts the final character of each word (enDstoPgotOgosuBinpuT...). Either way, shifted characters can be easily tested during a search, making this an efficient way to store data. Just be sure to remember that when we refer to the seventh bit, we're talking about the leftmost bit.

Springboard

In the 6502 chip instruction set, there aren't any instructions for giant branches. Some chips allow you to branch thousands of bytes away, but our chip limits us to 127 bytes in either direction from the location of the branch. Normally, this isn't much of a problem. You JSR or JMP when you want to go far away.

But as you assemble, you'll be making tests with BNE and BEQ and their cousins in the B group. Then, later, you'll add some more pieces of programming between the branch instruction and its target. Without realizing it, you'll have moved the target too far away from the branch instruction. It will be a branch out of range.

This is pretty harmless. When you assemble it, LADS will let you know. It will print a bold error message, print the offending line so you can see where it happened, and even ring a bell in case you're not paying attention. What can you do, though, when you have branched out of range? Use a springboard.

The easiest and best way to create a giant branch is this:

```
100 LDA 15
110 BEQ JTARGET
.
170 JTARGET JMP TARGET; THIS IS THE SPRINGBOARD
.
.
930 TARGET INY ; HERE IS OUR REAL DESTINATION FROM
                 LINE 110
```

When you get a BRANCH OUT OF RANGE ERROR message, just create a false target. In LADS, the letter J is added to the real target name to identify these springboards (see line 170 above). All a springboard does is sit somewhere near enough to the branch to be acceptable. All it does is JMP to the true target. It's like a little trampoline whose only purpose is to bounce the program to the true destination of the branch.

One final note: To make it easy to locate programming explanations in the text of this book, all line numbers are in boldface. Most of the chapters in the book cover a single major subprogram. At the end of a chapter is the appropriate source code listing. It is these listings to which the boldface line numbers refer.

Now, let's plunge into the interior of the LADS assembler. We'll start with the equate labels, the definitions of special addresses within the computer.

Chapter 2

Defs:

Equates and Definitions

Defs: Equates and Definitions

Let's get started. Recall that the boldface numbers within the text refer to line numbers within the program listings at the end of each chapter. The first section of LADS defines many of the variables which are used throughout the program. It's called "Defs."

Defs for Relocatability

One of the advantages of advanced assemblers, LADS included, is that they create object code (runnable ML programs) which are both *relocatable* anywhere within a computer's RAM memory as well as *transportable* between computer brands and models.

If you want to put LADS at $5000 instead of $2AF8, you can relocate it quite simply: Just change line 10 in the Defs source code file, the first file in the chain of LADS source code files. As written, line 10 reads *= 11000 (equivalent to *= $2AF8) and that causes the entire object program to start at that address. Changing line 10 to *= $5000 relocates LADS when you next assemble it. If you include the pseudo-op .D, the object program will be saved to disk under the filename you specify.

In the source code of LADS itself, at the end of this chapter, the ".D LADS64" in line 30 will create a version of LADS on disk by the name of LADS64 and if you later LOAD "LADS64",8,1 it will come into your computer ready to run with a SYS 11000. If you change the start address in line 10, however, to $5000, and then reassemble the source code, your LADS will start with a SYS 20480 (decimal for $5000).

The numbers generated by the assembly (the object code) will be sent to a disk file if you specify that with .D. They will be sent into RAM memory if you use the .O pseudo-op. If you do turn on storage of object code to memory, LADS will send the results of the assembly right into memory *during the assembly process*. This can cause mysterious difficulties unless you are careful not to assemble over LADS itself. If you have created a version of LADS which starts at $4C00 and you then start assembly of some object program at $5000, you'll eat into LADS itself. LADS is about 5K long. This, of course, would cause havoc. Using the .D pseudo-op is safe enough, since the new ML program assembles to disk. But the .O pseudo-op will send bytes right into RAM during assembly.

Be aware, too, that LADS builds its label array down from the start of its own code. During assembly, the labels and their values are stored in a growing list beneath the start address of LADS (where you SYS to start the assembler). If you send object code into an area of RAM which interferes with this array, you'll get lots of UNDEFINED LABEL errors. So be sure you know where you're putting object code if you store it in RAM during assembly by using the .O pseudo-op.

Defs for Transportability

The only part of LADS which is intensely computer-specific is this first file, this first subprogram, called Defs. Here we define all the machine-specific equates. (An *equate* is the same thing as a variable definition in BASIC. For example, RAMSTART = $2B is a typical equate.) We'll use the Commodore 64 Defs (Program 2-1) as our example. The labels (variable names like RAMSTART) for all other computers' versions of LADS will be the same—only the particular numbers assigned to these labels will vary. The addresses of pointers and ROM routines vary between computer models.

Defs contains the definitions of all zero page or ROM addresses that will be used in the rest of the source code. Once again, remember that *all zero page equates must be defined at the start of the source code* (Defs illustrates that rule: Defs is the first part of the LADS source code). From lines 60 to 170 we define the locations within zero page that we'll be using. In line 70 we define the top of the computer's RAM memory. We're going to lower it from its usual spot to fall just below where LADS itself starts.

ST is the location where errors in disk file manipulation can be detected. Like all of these zero page *equates,* this location varies from computer to computer. LOADFLAG (line 90) signals the computer that we want to LOAD a program file (rather than VERIFY a previously SAVEd program file). This flag will be set in the version of LADS which assembles from RAM memory (and LOADs in chained source code programs from disk). This RAM-based version of LADS will be created later in Chapter 11, the chapter on modifying LADS.

Disk I/O Information

The next five definitions show where information is stored just before a disk operation. They tell the operating system where in memory a filename is located, how long the name is, the file number, the file's secondary address, and the device number (8 for disk, 4 for printer, in Commodore computers).

CURPOS always contains the position of the cursor onscreen (as a number of spaces over from the left of the screen). We'll use this to format the screen listings. And the final machine-specific zero page definition is RAMSTART. It tells LADS where BASIC RAM memory *starts*. It, too, is used in the version of LADS which assembles from RAM.

Why do we need to define these locations if the operating system uses them? Because we're going to use a few of the built-in BASIC routines to handle the I/O (Input/Output) operations for us when we need to communicate with a peripheral. To OPEN a file, for example, we need to set up several of these pointers. To OPEN file #1, we have to put a 1 into address $B8 (that's where the file number is held on the Commodore 64). But why not just use LDA #1: STA $B8? Why do we want to use these labels, these variable names?

Programming with pure numbers instead of labels prevents transportability. It locks your program into your computer, your model. It's far easier to change this single equate in line 120 to $D2 to make the program run on a PET/CBM with BASIC 4.0 than it would be to go through the entire source code, changing all B8's to D2's. Also, if you buy a newer model and they've moved things around in zero page (they almost always do), making the adjustments will be simple. You just use a map of the new zero page and make a few changes in the Defs file.

LADS Zero

Because LADS needs to use the valuable Indirect Y addressing mode—LDA (12),Y or STA (155),Y—it will want to usurp a few of those scarce zero page locations itself. Line 170 defines a two-byte temporary register called TEMP which will be used in many ways. SA is going to function as a two-byte register for the LADS Program Counter which will keep track of where we are currently storing object bytes during the assembly process.

MEMTOP is used in the construction of our label data base. It will always know where the last symbol in our label table was stored. All through pass 1 it will be lowering itself, making room for new symbols and labels. (This data base will later be referenced as we fill in the blanks on pass 2.) PARRAY makes that search through the symbol table on pass 2 easy and fast. It points us through the array. PMEM is used as a pointer during assembly from RAM, if you decide to use the RAM-based version of LADS described in Chapter 11. The uses of all these variables will become clear when we examine, throughout the book, the techniques which utilize them.

Borrowing from BASIC

The next section, lines 190–320, defines the routines within BASIC ROM memory that we're going to use. Naturally, these are particular to each computer brand and model, so we want them up front where they can be easily identified and changed.

BASIC always has an entry point called the *warm start address*, a place where you can jump into it "warmly." But there's another entry that's not as gentle. Many BASICs clear out RAM memory and radically reset pointers, etc., when you first turn on the computer. This is called the *cold start* entry point, and it's as much of a shock to the computer as walking outdoors into a winter wind is to you. We don't want this shock when we return from LADS to BASIC. Instead, we want the RAM memory left alone. After all, LADS is in there and possibly an object or source program is in there too. So when assembly is finished, we want to go into BASIC via the *warm start* entry point.

KEYWDS is the address of the first BASIC keyword. We'll see why we need this address in the chapter on the Indisk subprogram. OUTNUM is a ROM routine which is used to print line numbers for the BASIC LIST command. We'll use it in a similar way to list the line numbers of our source code.

OPEN, CHKIN, CHKOUT, CLRCHN, and CLOSE allow us to communicate with the disk drives and printers. CHARIN is like BASIC's GET command, PRINT like PRINT. STOPKEY sees if you've pressed the STOP or BREAK key on your keyboard. And, last, SCREEN tells LADS where in RAM your video memory starts.

The use of these routines, and the ways that ML programs can borrow from BASIC, will be covered in detail as they appear in the LADS source files. For now, we only need to know that they are defined here, in Defs, and can be quickly changed to suit different computers, different BASICs.

There you have it. We'll be explaining these pointers and registers as we come upon them in the explication of LADS. Now on to the heart of LADS, the section which evaluates all the mnemonics (like LDA) and addressing modes and turns them into opcodes (like A9) that are the machine's language. This next section, Eval, is—by itself—a complete assembler. It would stand alone. The rest of the sections of LADS add things to this core, things like disk management, arithmetic and other pseudo-op routines, label interpretation, screen and other output, and a host of other niceties. But Eval is the sun; the rest of the routines are lesser bodies, planets in orbit around it.

Note: Because the Defs subprogram is computer-specific, there are five source code listings at the end of this chapter, one for each computer. There are also multiple listings in Chapter 5 since it deals with computer-specific peripheral communication. However, the majority of chapters will have only a single complete listing, followed by the few modifications required by the different computers, because the majority of LADS' source code is identical and entirely transportable between 6502-based computers.

Program 2-1. Defs: Commodore 64

```
10 *= 11000
20 .NO
30 .D LADS64
40 ; "DEFS64"  EQUATES AND DEFINITIONS FOR COMMODORE 64
50 ;--------------------- MACHINE SPECIFIC ZERO PAGE EQUATES -----------
60 RAMSTART = $2B; BASIC'S START OF RAM MEMORY POINTER
70 BMEMTOP = $37; BASIC'S TOP OF RAM MEMORY POINTER
80 ST = 144; STATUS WORD FOR DISK/TAPE I/O
90 LOADFLAG = $93; FLAG WHICH DECIDES LOAD OR VERIFY (0 = LOAD)
100 FNAMELEN = $B7; LENGTH OF FILENAME FOR OPEN A FILE
110 FNAMEPTR = $BB; POINTER TO FILENAME LOCATION IN RAM.
120 FNUM = $B8; CURRENT FILE NUMBER FOR OPEN, GET & PUT CHARS TO DEVICE
130 FSECOND = $B9; CURRENT SECONDARY ADDRESS FOR OPEN
140 FDEV = $BA; DEVICE NUMBER (8 FOR COMMODORE DISK)
150 CURPOS = 211; POSITION OF CURSOR ON A GIVEN SCREEN LINE.
160 ;--------------------- LADS INTERNAL ZERO PAGE EQUATES -----------
170 TEMP = $FB:SA = $FD:MEMTOP = $B0:PARRAY = $B2:PMEM = $A7
180 ;------------------ MACHINE SPECIFIC ROM EQUATES -----------
190 BABUF = $0200; BASIC'S INPUT BUFFER
200 TOBASIC = $A474; GO BACK TO BASIC
210 KEYWDS = $A09E; START OF KEYWORD TABLE IN BASIC
220 OUTNUM = $BDCD; PRINTS OUT A (MSB), X (LSB) NUMBER
230 OPEN = $E1C1; OPENS A FILE  (3 BYTES PAST NORMAL OPEN IN ROM).
240 CHKIN = $FFC6; OPENS A CHANNEL FOR READ (FILE# IN X)
250 CHKOUT = $FFC9; OPENS CHANNEL FOR WRITE (FILE# IN X)
260 CHARIN = $FFE4; PULLS IN ONE BYTE
270 PRINT = $FFD2; SENDS OUT ONE BYTE
280 LOAD = $E175; LOAD A BASIC PROGRAM FILE (SOURCE CODE FILE) INTO RAM.
281 ; (F322 FOR UPGRADE/E172 FOR VIC)
```

```
290 CLRCHN = $FFCC; RESTORES DEFAULT I/O
300 CLOSE = $FFC3; CLOSE FILE (FILE# IN A)
310 STOPKEY = $FFE1; TESTS STOP KEY, RETURNS TO BASIC IF PRESSED.
320 SCREEN = $0400; ADDRESS OF 1ST BYTE OF SCREEN RAM
330 ;------------------
340 .FILE EVAL
```

Program 2-2. Defs: VIC-20

```
10 *= 11000
20 .D LADSV
30 .NO
40 ;   VIC VERSION
50 ; "DEFSV"  EQUATES AND DEFINITIONS
60 ;---------------------  MACHINE SPECIFIC ZERO PAGE EQUATES ----------
70 BMEMTOP = $37; BASIC'S TOP OF MEMORY POINTER
80 ST = 144; STATUS WORD FOR DISK/TAPE I/O
85 LOADFLAG = $93
90 FNAMELEN = $B7; LENGTH OF FILENAME FOR OPEN A FILE
95 FNAMEPTR = $BB; POINTER TO FILENAME LOCATION IN RAM.
100 FNUM = $B8; CURRENT FILE NUMBER FOR OPEN, GET & PUT CHARS TO DEVICE
110 FSECOND = $B9; CURRENT SECONDARY ADDRESS FOR OPEN
120 FDEV = $BA; CURRENT DEVICE NUMBER
130 CURPOS = 211; POSITION OF CURSOR ON A GIVEN SCREEN LINE.
135 RAMSTART = $2B; POINTER TO START OF RAM MEMORY (FOR RAM-BASED ASSEM.)
140 ;---------------------  LADS INTERNAL ZERO PAGE EQUATES ----------
150 TEMP = $FB:SA = $FD:MEMTOP = $B0:PARRAY = $B2:PMEM = $A7
160 ;------------------  MACHINE SPECIFIC ROM EQUATES ----------
170 TOBASIC = $C474; GO BACK TO BASIC
175 BABUF = $0200; BASIC'S INPUT BUFFER
176 LOAD = $E172
```

```
180 KEYWDS = $C09E; START OF KEYWORD TABLE IN BASIC
190 OUTNUM = $DDCD; PRINTS OUT A (MSB), X (LSB) NUMBER
200 OPEN = $E1BE; OPENS A FILE  (3 BYTES PAST NORMAL OPEN IN ROM)
210 CHKIN = $FFC6; OPENS A CHANNEL FOR READ (FILE# IN X)
220 CHKOUT = $FFC9; OPENS CHANNEL FOR WRITE (FILE# IN X)
230 CHARIN = $FFE4; PULLS IN ONE BYTE
240 PRINT = $FFD2; SENDS OUT ONE BYTE
250 CLRCHN = $FFCC; RESTORES DEFAULT I/O
260 CLOSE = $FFC3; CLOSE FILE (FILE# IN A)
270 STOPKEY = $FFE1; TESTS STOP KEY, RETURNS TO BASIC IF PRESSED.
280 SCREEN = $1000; ADDRESS OF 1ST BYTE OF SCREEN RAM (W/EXPANDED MEMORY)
640 .FILE EVAL
```

Program 2-3. Defs: PET/CBM 4.0 BASIC

```
10 *= 11000
20 .NO
30 .D LADS
40 ; "DEFS"  EQUATES AND DEFINITIONS FOR PET/CBM 4.0 BASIC
50 ;--------------------- MACHINE SPECIFIC ZERO PAGE EQUATES -----------
60 RAMSTART = $28; BASIC'S START OF RAM MEMORY POINTER
70 BMEMTOP = $34; BASIC'S TOP OF RAM MEMORY POINTER
80 ST = 150; STATUS WORD FOR DISK/TAPE I/O
90 LOADFLAG = $9D; FLAG WHICH DECIDES LOAD OR VERIFY (0 = LOAD)
100 FNAMELEN = $D1; LENGTH OF FILENAME FOR OPEN A FILE
110 FNAMEPTR = $DA; POINTER TO FILENAME LOCATION IN RAM.
120 FNUM = $D2; CURRENT FILE NUMBER FOR OPEN, GET & PUT CHARS TO DEVICE
130 FSECOND = $D3; CURRENT SECONDARY ADDRESS FOR OPEN
140 FDEV = $D4; DEVICE NUMBER (8 FOR COMMODORE DISK)
150 CURPOS = 198; POSITION OF CURSOR ON A GIVEN SCREEN LINE.
160 ;--------------------- LADS INTERNAL ZERO PAGE EQUATES -----------
```

```
170 TEMP = $FB:SA = $FD:MEMTOP = $BB:PARRAY = $BD:PMEM = $BF
180 ;------------------- MACHINE SPECIFIC ROM EQUATES -----------
190 BABUF = $0200; BASIC'S INPUT BUFFER
200 TOBASIC = $B3FF; GO BACK TO BASIC
210 KEYWDS = $B0B2; START OF KEYWORD TABLE IN BASIC
220 OUTNUM = $CF83; PRINTS OUT A (MSB), X (LSB) NUMBER
230 OPEN = $F563; OPENS A FILE  (3 BYTES PAST NORMAL OPEN IN ROM).
240 CHKIN = $FFC6; OPENS A CHANNEL FOR READ (FILE# IN X)
250 CHKOUT = $FFC9; OPENS CHANNEL FOR WRITE (FILE# IN X)
260 CHARIN = $FFE4; PULLS IN ONE BYTE
270 PRINT = $FFD2; SENDS OUT ONE BYTE
280 LOAD = $F356; LOAD A BASIC PROGRAM FILE (SOURCE CODE FILE) INTO RAM.
281 ; (F322 FOR UPGRADE/E172 FOR VIC/E175 FOR 64)
290 CLRCHN = $FFCC; RESTORES DEFAULT I/O
300 CLOSE = $F2E2; CLOSE FILE (FILE# IN A)
310 STOPKEY = $FFE1; TESTS STOP KEY, RETURNS TO BASIC IF PRESSED.
320 SCREEN = $8000; ADDRESS OF 1ST BYTE OF SCREEN RAM
330 ;------------------
340 .FILE EVAL
```

Program 2-4. Defs: Apple

```
10 *= $79FD
20 .D LADS
30 .NO
40 ;APPLE VERSION
50 ; "DEFS"  EQUATES AND DEFINITIONS
60 ;--------------------- MACHINE SPECIFIC ZERO PAGE EQUATES -----------
70 BMEMTOP = $4C; BASIC'S TOP OF MEMORY POINTER
80 TXTPTR = $B8; POINTER TO NEXT BYTE OF TEXT
85 FNAMELEN = $F9; LENGTH OF FILE NAME
```

```
90 CHRGET = $B1; GET NEXT BYTE OF TEXT
95 PRGEND = $AF; POINTER TO END OF PROGRAM
100 HIGHDS = $94; HIGH DESTINATION OF BLOCK TRANSFER UTILITY (BLTU)
110 VARTAB = $69; VARIABLE TABLE POINTER
130 CURPOS = 36; POSITION OF CURSOR ON A GIVEN SCREEN LINE.
140 ;--------------------  LADS INTERNAL ZERO PAGE EQUATES -----------
150 TEMP = $FB:SA = $FD:MEMTOP = $EB:PARRAY = $ED
155 PARM = $2A:FMOP = $2C
160 ;-----------------  MACHINE SPECIFIC ROM EQUATES -----------
170 TOBASIC = $3D0; GO BACK TO BASIC
175 BABUF = $0200; BASIC'S INPUT BUFFER
180 KEYWDS = $D0D0; START OF KEYWORD TABLE IN BASIC
190 OUTNUM = $ED24; PRINTS OUT A (MSB), X (LSB) NUMBER
200 CSWD = $AA53; ADDRESS OF CHARACTER OUTPUT ROUTINE
210 COUT = $FDF0; OUTPUT ONE BYTE
220 PRNTR = $C090; I/O LOCATION FOR PRINTER
230 PRNTRDN = $C1C1; PRINTER READY SIGNAL
240 LINGET = $DA0C; GET LINE NUMBER FROM TXTPTR INTO LINNUM
250 LININS = $D46A; INSERT BASIC LINE INTO BASIC TEXT
280 SCREEN = $0400; ADDRESS OF 1ST BYTE OF SCREEN RAM
640 .FILE EVAL
```

Program 2-5. Defs: Atari

```
100 *= $8000
110 .D D:LADS.OBJ
120 ST = $01
130 FNAMELEN = $80
140 FNAMEPTR = $81
150 FNUM = $83
160 FSECOND = $84
170 FDEV = $85
180 CURPOS = 85
190 TEMP = $86
200 SA = $88
210 MEMTOP = $8A
220 PARRAY = $8C
230 INFILE = $8E
240 OUTFILE = $8F
250 PMEM = $A0
260 RAMFLAG = $A2
270 BABUF = $0500
280 SAVMSC = $58
290 .FILE D:EVAL.SRC
```

Chapter 3

Eval:

The Main Loop

Eval: The Main Loop

Eval is the heart of LADS. It is the main loop. It starts assembly at START (line **30**) and ends assembly at FINI (line **4250**). Throughout Eval, JSRs take us away from the main loop to perform various other tasks, but like mailmen, all the other routines in the assembler start out from Eval, the post office, and they all RTS back to it when their work is done.

For convenience, references to lines within the source code listing at the end of the chapter are boldface inside parentheses. Also, to distinguish label names like FINI from the names of one of the 13 sections of LADS (a subprogram like Eval), we'll put label names in all caps, but just capitalize the first letter of the subprograms of the assembler.

Preliminaries, Preparations

Most programs have a brief *initialization* phase, a series of steps which have to be taken to fix things up before the real action of the program can commence. Variables have to be set to zero, files sometimes have to be opened on a disk, *defaults* have to be announced to the program. (Defaults are those things a program will do unless you specifically tell it not to. A game might default to single-player mode unless you do something which tells it that there are two of you playing. LADS defaults to hexadecimal numbers for printer or screen listings and turns off all its other options.)

At its START, LADS loads the Accumulator with zero and runs down through 48 bytes of registers, flags, and pointers, stuffing a zero into each one. These flags are all needed by LADS to keep track of such things as which pass it's on, whether or not you want a printer listing, or want the results of an assembly to POKE into memory, or whatever. This initialization fills them all with zero. The label OP is the highest of these registers in memory, so we LDY with 48 and DEY down through them (see line **30**).

Let's take a minute to briefly review our terminology:

Register usually refers to the Accumulator (A), or the X or Y Register in the 6502 chip. It can also mean a single byte set aside to temporarily hold something. It's like a tiny *buffer*.

A *buffer* is a group of continuous bytes used to hold information temporarily. An input buffer, for example, holds the bytes you type in from the keyboard so they can be interpreted by BASIC. The bytes stay there until you type RETURN, BASIC stores the information into your program, and you type a new line into the input buffer.

A *flag* is a byte which is either on or off (contains either zero or some number) and signifies a "do it" or "don't do it," yes or no, condition. Of course, a single byte could hold a number of flags because each bit could be on or off. In fact, the Status Register in the 6502 chip does just that—it's only a single byte, but its bits are flags tested by CMP and the BNE, BEQ-type instructions. When you need a flag, though, it's easier to just use a whole byte and test it for zero or not-zero. An example of a flag in LADS is the PRINTFLAG. If nonzero, the assembler sends a printout of the assembly process to a printer. If zero, the printer remains silent and still. You *set* (turn on) the print flag with the pseudo-op .P; otherwise, the default is no printing.

A *pointer* holds a two-byte address. Many times pointers are put into zero page so they can be used by Indirect Y addressing: LDA ($FB), Y gets the byte from the address held in $FB and $FC (seen as a single, two-byte-long number). If

```
00FB 00
00FC 15
```

(remember that the 6502 expects these numbers to be backward; this two-byte group means $1500) then LDA ($FB),Y will load the A register (the Accumulator) with whatever byte is currently in address $1500. We can set up our own pointers. If they're not in zero page, they're likely holding some important address which a program needs to remember. In LADS, ARRAYTOP is such a non-zero-page pointer; it tells LADS where to start looking through the label table for a match. We'll look into this when we get to the subprogram Arrays.

Cleaning the Variables

At its start LADS must initialize its variables. If we didn't fill them with zero, there could be some other number in these bytes when we fire up LADS and that could cause unpredictable results. Then (**80**) we get the low byte of the start of LADS (using the pseudo-op #<START) and put it in the low

byte of MEMTOP (used by the Equate subprogram). We also put it into the pointer BASIC uses to show how much RAM memory it has available, BMEMTOP (line 70 in Defs). And, finally, put it in ARRAYTOP. ARRAYTOP will show where the LADS' data base of labels starts in memory (it builds downward from the location of LADS).

Then we take the high byte of START and put it into the high bytes of these three pointers.

Now for the defaults. There is only one. We want listings to be in hexadecimal unless we specifically direct the assembler otherwise with the .NH, no hex, pseudo-op. So we put #1 into the HXFLAG. The rest of the flags are left at zero. If you want different defaults, put #1 into some of the other flags. For example, if you usually want to watch the results on screen during an assembly, just create a new line: 185 STA SFLAG. This will cause a screen disassembly every time you use LADS. Putting this default into LADS itself merely saves you the time of adding the .S pseudo-op if you generally do want to watch the assembly onscreen. That does slow up the assembler, but with shorter programs, you might not notice the difference.

Where's the Source File?

LADS needs to know what you want to assemble. If you're using the RAM-based version of LADS (see Chapter 11), there's no need to give a filename to LADS; just SYS, and LADS will assemble what's already in RAM. But if you're in the normal LADS mode, assembling from a disk file, you'll have to announce which file. LADS looks at the upper left-hand corner of the screen to read the filename (**190**). If it finds a space #32, it checks for another space (**310**) before giving up. This way you can have continuous names like FILENAME as well as two-word names like FILE NAME. Whatever it finds onscreen, it stores in the buffer FILEN. It also takes care of characters which are below the alphabet in the ASCII code by adding 64 to them if they fall below 32 (**240**). The Atari version asks for the filename from the keyboard in the manner of a BASIC INPUT command.

When the filename is stored in the buffer, we JSR to Open1, the subprogram which handles all I/O, all communication with peripherals. In this case, communication will be with the disk drive.

After the file is opened for reading, we JSR to another subprogram, Getsa, the get-start-address routine. It just looks for *= (the start address pseudo-op) and, finding it, returns to Eval where the number following that symbol will be evaluated. If it doesn't find a *=, that can only mean two things. Either there is no program on the disk by the name you put onscreen or LADS did find the program, but no starting address was given as the first item in the source code. Both of these situations are capable of driving LADS insane, so Getsa aborts back to the safety of BASIC after leaving you a message onscreen.

This SMORE routine (**370**) will be used again when we've completed the first pass of the assembly process. The first pass goes through the entire source file, storing all the names of the labels and their numeric values into an array.

When we finish making this collection of labels, our label array, we've got to make a second pass, filling in the opcodes and replacing those labels with numbers. It's here, at SMORE, that we jump to start the second pass.

A zero is given to ENDFLAG to keep the assembler running. If the ENDFLAG is left up, is not zero, the assembler assumes it has finished its job and stops.

The initialization is completed with a JSR to the subprogram Indisk which pulls in the number you wrote as the starting address following *=. This number is left in LADS' main input buffer called LABEL. Before dealing with this number, though, we check to see if we're on the first pass (**410**) and, if so, print the word LADS onscreen after a JSR PRNTCR which prints a carriage return. Routines beginning with PRNT like PRNTSPACE and PRNTLINE are all grouped together in the subprogram Findmn. They're used by most of the subprograms and print various things to the printer or screen.

Now we need to put the starting address into the pointer SA which always holds the current target for any of our assembled code during execution. If the HEXFLAG is up, that means you wrote something like *= $5000 and hex numbers are translated by the subprogram Indisk before it RTSs back to Eval. Decimal numbers like *= 8000, however, are not translated into the two-byte integers that ML (machine language) works with, so we need to send decimal numbers to Valdec (another subprogram) to be turned into ML integers (**610**). The pointer called TEMP is made to point to LABEL so Valdec will know where to look for the number.

It's important to realize that numbers coming in from the disk or from RAM memory are in ASCII code, as *characters*, not true integer numbers. That is, the characters in a number like 5000 will come into the LABEL buffer as they appear in RAM or on a disk file. 5000 would be (in hexadecimal notation) 35 30 30 30; these are the character codes for 5-0-0-0. It's Valdec's job to transform this into 00 50, an ML integer. When we get to Valdec, we'll see just how this is done. It's a useful technique to learn since any numbers input from a keyboard will also be in this ASCII form and will need to be massaged a bit before they'll make sense to ML.

Remembering the Start Address

When, at STAR1, we finally have an ML integer in the little two-byte variable called RESULT, we can transfer the integer to SA. And we put the integer into the variable TA, too, so that we'll have a permanent record of the starting address. SA will be *dynamic*; it will be changing throughout assembly to keep track of the current assembly address. It will be LADS' Program Counter. TA will always remember the original starting address.

By this time you might be thinking that all this is hard to follow. TA and RESULT and LABEL don't mean much at this point. We've plunged into Eval, the most condensed, the most intensive, section of the entire program. As the main loop, Eval will send tasks to be accomplished to many subroutines, in subprograms which we've not yet examined. It's like landing in a strange city without a map. You see street signs, but they mean nothing to you yet. But this is one of the best ways to learn if you can be patient and ignore the temporary gaps in your knowledge and the momentary sensations of confusion.

We're gradually building a vocabulary and mapping out some of the pathways which make up the language LADS and the ways the ML works. The subprograms are, by and large, easier to follow. They're more self-contained. But bear with this tour through Eval. It makes what follows easier to grasp and offers a foundation—however unconscious at this point—for a deeper appreciation of the ways that ML does its magic.

The Main Routine

Every line of source code which LADS examines begins with STARTLINE (**690**). The ML between STARTLINE and P (**5520**) is, in effect, an assembler. The rest of the routines and subprograms deal with the niceties, the auxiliary efforts of the assembler—pseudo-ops, built-in arithmetic routines, I/O, printout formatting, and so forth.

In fact, this section of LADS is based on the BASIC assembler, the Simple Assembler, from my previous book, *Machine Language for Beginners*. If you want to see how a large BASIC program can be translated into ML, you might want to compare the Simple Assembler to the rest of Eval. There are some comments within the listing of LADS' source code which refer to the BASIC lines within the Simple Assembler (see lines 3270 and 3410 for examples), and a number of the labels, starting at **4670**, also refer to their BASIC line number equivalents in the Simple Assembler. L680 is a label to LADS, but is also a reference to an equivalent line, 680, in the BASIC of the Simple Assembler.

It's LADS' job to take each line in the source code and translate it into runnable ML object code. LADS would take the source line 10 LDA #15 and change the LDA into 169 and leave the 15 as 15. The value 169 is the ML opcode for the Immediate addressing mode of LoaDing the Accumulator. Then LADS would send these two bytes of object code, 169 15, to any of four places depending on what destinations you had specified as pseudo-ops in the source code. The .D pseudo-op would send 169 15 to a disk file, .P to the printer, .S to the screen, and .O directly into RAM memory.

When LADS first looks at at each source code line, STARTLINE checks the ENDFLAG to be sure it's safe to continue. If ENDFLAG is zero, we BEQ to the JSR to Indisk. (Otherwise, the program would go down to FINI and close up shop, its work finished.)

Indisk is the second largest subprogram, and LADS will be gone from Eval a long time by the computer's sense of time. For us, this detour happens in a flash, and a lot happens. Indisk can even JSR into other subprograms, but we'll see that in a later chapter. All we need to realize now is that each source line needs to be pulled onto our examination desk so LADS can pick it apart and know what to assemble.

Our examination desk is the buffer called LABEL. First a line of source code is laid out on the desk. To prepare for the exam, we put down the EXPRESSF(lag) and the BUFLAG, although they might be raised again during the evaluation to come. EXPRESSF tells LADS whether the expression following a mnemonic like LDA is a label or a number. It signals the difference between LDA SPRITE and LDA 15. BUFLAG tells whether or not there is a REM-like comment attached to the line under examination. If there is a comment, we'll want the assembler to ignore the remarks, but the screen or printer should nevertheless display them.

Now, as we often will, we check PASS (**760**) to see if it's the first or second time through the source code. On the first pass, we're not going to print things to a printer or the screen, so we'd jump to MOE4 and ignore the next series of printouts.

But if it's the second pass, we check the SFLAG, the screen flag, to find out if we should print to the screen. If the answer is yes, we print a line number, a space, the SA (current address), and another space. Don't worry about LOCFLAG just yet.

Now we want to know if there's any math to do. PLUSFLAG is up when the line contains something like this: LDA SCREEN+5. If it does, we briefly detour to the subprogram Math to replace SCREEN+5 with the correct, calculated number.

The Inner Core

Now we're at the true center, the hot core, of LADS: Line 900 is the pivot around which the entire structure revolves. This JMP to Findmn accomplishes several important things and sets up the correct pathways for the assembler to follow in the future. Findmn finds a mnemonic. Say LADS is examining this line:

10 LDA 15

After Findmn does its job and JMPs back to Eval, there would be a 1 in the TP register (it's like a BASIC variable, called TP for "type"). And there would be a 161 in the OP, for opcode, register.

That 161 is not the number we'll want POKEd into memory. 161 *is* the right number for the LDA (something,X) addressing mode, but it's wrong for the other modes, including LDA 15. Nevertheless, any LDA will first get a 161, the base opcode. It's the lowest possible opcode for an LDA; the other LDA addressing modes can be calculated by adding to 161. LDA 15 is Zero Page addressing and its opcode is 165. Eval's main job is to start off with the lowest, the base opcode for a particular mnemonic like LDA, and then make adjustments to it when the correct addressing mode is detected. Eval establishes the addressing mode when it examines the line and looks for things like the # symbol and so forth. As we'll see, this examination will modify the OP number until the correct opcode is calculated.

For now, though, it's enough that we return from Findmn with a base opcode number, something reliable to work from, stored in the variable OP. By the way, Findmn gets these numbers, TP and OP, from a table in the subprogram Tables. We'll look at it at the very end of our exploration of LADS in Chapter 9. Tables is where all the constants are stored.

When No Match Is Found

Sometimes Findmn won't find a match when it looks through the table of mnemonics in the subprogram Tables. This means that the first word in the line under examination was *not a mnemonic*. If this happens, Findmn returns (via a JMP) back into Eval where labels are analyzed. Eval then knows that this first word isn't one of the 6502 commands. Instead, it must be a label.

Labels in this first position in a line can be of two types: *address labels* and *equate labels*. An address label identifies a location within the program that will be the target for branches, jumps, JSR, etc. It's like giving names to subroutines so you could later JSR PRINTROUTINE. Here's an example:

100 START LDA #0

After the assembler finishes assembling this, we'll have:

100 3A00 A9 00 START LDA #0

The OP 161 has been changed to 169 (the hex number A9 in the example above), and we'll see how that was arrived at presently. But START has had no visible effect. It's just listed there, but doesn't affect the A9 or 00. START is a place marker. It hasn't been ignored. During the first pass, LADS stored START in an array along with the 3A00 address. That's why START can be called an *address label*. This is very much the way that BASIC reads a variable name, sticks it in an array, and puts the value of the variable up there with the name.

On pass 2, when all these labels are needed, the correct address will be there, waiting in the array. If LADS comes across a JSR START or a BEQ START, it will be able to search the array and replace the word START with the right number, the address.

The other possible kind of label is the *equate label.* It looks like this:

1100 SCREEN = $0400

It, too, is stored during the first pass and looked up during the second pass. But the equals sign shows that we should remember the value on the other side of the = symbol, not the address of the location of the label. In this example, whenever we want to store something onscreen, we don't need to calculate the correct address. $0400 is the first byte in screen memory (on the Commodore 64 in this example). So we can just STA SCREEN to put whatever is in A into the upper left-hand corner of the screen. Or STA SCREEN+200, or STA SCREEN+400, or whatever. (Adding numbers to SCREEN will, in this case, position our A lower on the screen.)

It's here that we decide whether we're dealing with one of the labels or with an ordinary mnemonic. If we JMP back from Findmn to EVAR (**920**), the first thing on the source code line was a mnemonic. If we JMP back from Findmn to EQLABEL, it wasn't a mnemonic (hence it's a label). EVAR evaluates the *argument,* the 15 in LDA 15. EQLABEL evaluates the other kind of *argument,* the label SPRITE in LDA SPRITE.

Simple and Other Types

Some of the mnemonics are quite straightforward. They've got no argument at all: INY, ROL, CLC, DEC, BRK, RTS, etc. There's no argument to figure out, and all of these self-contained instructions have the same addressing mode, *Implied addressing*. Fully 25 of the 56 mnemonics are of this type. We've called them type 0 (see the chapter on the Tables subprogram for an explanation of the types), and so Findmn puts a 0 into the TP variable. Our first step in the evaluation of any argument (**920**) is to check the TP, and if it's 0, go to the type 1 (meaning only one byte, the opcode itself) area. There, the single byte will be POKEd and printed if you've requested that with your pseudo-ops. And then we can go on to fetch a new line.

If it's a more complicated addressing mode, though, we continue evaluating, comparing it to type 3 (**940**). If you want, you can look up the mnemonics and the parallel types and ops tables in the Tables subprogram. Type 3's are the bit-moving instructions ROL, LSR, ROR, and LSR. They have a pattern of possible addressing modes in common. (It's this common pattern of addressing modes which underlies these *types*. They share the same potential addressing modes and can be evaluated and adjusted as a category rather than individually.)

In any case, we turn them into type 1 and then look at the fourth position in the storage buffer LABEL. If we could peer into this buffer, we might see either:

ASL

or

ASL 1500

That bare ASL is *not* an implied address like INY and CLC and the rest of those self-contained instructions we discussed above. These bit-moving instructions (ASL, ROR, etc.) are just like type 1 (LDA, etc.) with this single exception: They can have a special addressing mode all their own called *Accumulator* addressing. It's a rare one. In this mode, ASL would Arithmetic-Shift-Left the number in A, the Accumulator.

The point to grasp here is that, rare as a nude ASL is, we've got to include it in the assembler. So we check to see if there is a zero in the fourth position in our buffer, LDA LABEL+3. A zero means end-of-line. So we can detect from a zero that there is no argument and, hence, this is a case of *Accumulator addressing*. If it is, we need to add 8 to the base opcode for these bit-movers and then jump to the type 1 exit. If it isn't, we've already turned it into a type 1 (**970**) and from here on, we'll treat it as a member of that family. In effect, type 1's can have several addressing modes, so we must evaluate the mode. We go to EVGO.

Fat Y Loops

Before entering most ML loops, you'll first LDY #0. Y often functions as a counter, so it's set to zero, and then INY occurs at the *end* of the loop. But some loops require that we INY at the start or at least early within the loop. In such cases, we must LDY #255 before entering the loop. The first event within the loop is an INY, so in effect, Y becomes 0 right off the bat. When you increment 255, you get a zero.

EQLABEL is where we determine what kind of label we're dealing with. On the first pass, we don't care. All labels must be stored in our label table array for later reference on pass 2. On pass 2, though, we must go through the test in EVX1 (**1090**). And it's one of those fat Y loops that start off with a bloated Y Register. We put 255 into Y at the start.

We load the first character in the LABEL buffer. If it's zero (end of the line), there wasn't any argument. There should have been. This is a mistake. By this time, there *has* to be an argument. We've already eliminated the only addressing types that have no argument: Implied (type 0) and Accumulator (a variant of type 3). If there's no argument, the source code is defective. There should be an argument. We've got to print an error message.

NOAR is tucked away at line 520 of the Equate subprogram. We'll get to it later. It just prints a "no argument" error message. But we should clear up the little mystery surrounding the bounce we just took. We BEQ GONOAR (**1110**) only to JSR NOAR (**1320**). Why? This is one of those springboards we discussed in Chapter 1.

The B instructions, the branchers like BEQ, can move us only 127 bytes in either direction, forward or backward, from their location. This is sometimes not far enough. LADS will alert you to this if you should try to branch further than you can. It will print BRANCH OUT OF RANGE and ring the bell. The easiest solution to this problem is to simply have the branch go to a nearby JMP or JSR. *They* can fly off to any address in the computer. Have them act as springboards, bouncing you to your actual target.

The alternative is to move your target closer to the branch. The target is probably a subroutine. But moving a subroutine is often a lot more trouble than simply creating a springboard.

Back to the evaluation (**1120**). If there is an argument, we move it up to another buffer called FILEN. Then we check for the blank character, 32, before leaving this loop. The label name gets moved up to FILEN for further analysis. Then we INY and look at the next character.

Which Kind of Label?

If the first thing after a blank character is =, we've got an equate label like:

100 NAME = $500

If it is an equate label, we ignore it because we're on the second pass here. Line 330 sends us over this section if it's the first pass. There's no need to pay any attention to equate labels on the second pass, so we jump to INLINE, the preparations for getting a new line to evaluate.

But it might be the other type of label, an address label like:

100 START LDA #15

On pass 2 we can also ignore START, the label part of this line. Both types of labels have already been safely stored in our array during pass 1. Nevertheless, following the address-type label is some code we cannot ignore. On pass 2 LADS must assemble that LDA #15.

NOTEQ (not equate type) moves the address label up to a buffer called FILEN while at the same time moving the LDA #15 over to the start of the LABEL buffer. It's doing two things at once. This is how these buffers look before NOTEQ **(1180–1200)**:

LABEL START LDA #1500000000000
FILEN 000000000000000000000000

and after NOTEQ:

LABEL LDA #150A #1500000000000
FILEN START0000000000000000000

START is up at FILEN and can be printed out later for a listing. But what good is that mess in the LABEL buffer? It will work perfectly well because that 0 in the eighth position is the *delimiter*. It tells LADS to ignore any random characters following it. Remember that these numbers are stored in memory as ASCII code, not as literal numbers. 15 would be stored as 49 53. 150 (the number 150) would be stored as 49 53 48. But a different kind of 150, where that final 0 is a true zero, a delimiter, would be stored as 49 53 0. So when we go to look at and assemble the information in LABEL, LADS will only work with LDA #15 and ignore the 0A #150000, etc., the remnants of the old line. All is now ready for the assembler to take a look at a mnemonic and its argument, so we JMP to MOE4 (**1310**). If this had been pass 1, we would have bypassed all this and leapt from 1070 right down to 1330, where we go to the subprogram Equate, which stores labels and their values in the label table array. But both pass 1 and pass 2 must continue to work out the addressing modes by going to MOE4. Why should we need to worry about addressing modes on pass 1 since LADS doesn't POKE anything into memory or save anything to disk during pass 1?

LADS must keep an accurate PC (Program Counter) during pass 1 to know what value to assign to address type labels. Otherwise, the address labels would be inaccurate:

```
10 START INC 15
20 LDA 15
30 BEQ FINISH
40 JMP START
50 FINISH RTS
```

Notice that both INC 15 and LDA 15 are Zero Page addressing. They occupy two bytes in memory. But they could have been Absolute (LDA 1500) addressing, or other modes which use up three bytes. LADS has no way of knowing, by reading LDA or INC alone, whether to raise the program counter by two or by three. All this wouldn't matter much except for that label FINISH in line 50. It has to be assigned its proper address *during pass 1* and stored in the array. That means LADS needs to know exactly how many bytes it is from START to FINISH.

Consequently, LADS has to check out the arguments of INC and LDA to see whether they're addressing modes using up two or three bytes. This Program Counter is kept in a variable in LADS called SA. It's constantly changing during both passes of the assembly, but it is used during pass 1 to assign numbers to address labels like START and FINISH.

We'll deal with the next routine, EVEXLAB (**1360**), shortly. Let's go first to MOE4 and see how LADS analyzes arguments.

We've Been Here Before

Recognize MOE4 (**900**)? We already discussed it. It JSRs to FINDMN and JMPs back to EVAR (**920**) having recognized a

6502 mnemonic or JMPs to evaluate a label if it didn't recognize a mnemonic. In our example, it will find LDA #15 this time, JMP to EVAR, and end up going to EVGO (from **950**).

Here at EVGO, LADS has to decide whether it's dealing with a normal numeric argument like #15 or an *expression label,* a word like SOUND. Imagine that we'd started off by defining the label SOUND:

10 SOUND = 15

When we later wanted to indicate 15, we could substitute the word (LDA #SOUND) for the number (LDA #15).

EVGO distinguishes labels from numbers by using the ASCII code. In this code, letters of the alphabet have a numeric value 65 (the letter A) and go up from there. Thus, if the character in the fourth position (see line **1490**) is less than 65, if it triggers a BCC, we don't raise the EXPRESSF(lag). That flag indicates a nonnumeric expression. In other words, the expression has a letter of the alphabet so it must be a label. Similarly, EVMO2A raises the Y offset and tests the fifth character. If it's a zero, we've got a single-letter label, like P (**1540**). Meanwhile, we're moving the label up to a buffer called BUFFER. And, again, we check for a character with a value lower than 65.

EVMO2 (1600) continues to move the label from one buffer (LABEL) to another (BUFFER). It only stops when it finds a zero indicating the end of the line. Note that both number expressions (arguments) like #15 *as well as* label expressions like #STOOL are moved from the LABEL buffer up to the BUFFER buffer. The only distinction between them is signaled by the raising of the EXPRESSF(lag) when there's a label rather than a number. For numbers, EXPRESS stays down, stays 0.

Hex Numbers Are Already Evaluated

EVMO3 (**1660**) puts the label's size, the number of characters in the label, into the variable ARGSIZE and checks to see if the HEXFLAG is up. The HEXFLAG is sent up in the subprogram Indisk if a $ symbol is noticed as a line is streaming into LADS. So if HEXFLAG is BNE, not equal to zero, it's up and we can jump right down to L340, which starts to figure out the addressing mode. If the EXPRESSF is up, that means a word label, not a number, so we have to go to EVEXLAB to get the number to substitute for the label. Otherwise, we've got a decimal number to work with as our argument (**1730**).

The whole function of lines 1730–1840 is to have the variable TEMP pointing to the first ASCII number in the label. That's why we keep INCrementing TEMP until we point to a character that is not BCC, less than the 0 ASCII character (48) in line 1830. Then we have to test for the (left parenthesis or , comma character. If it is one of them, it can put in a true zero as a delimiter.

When the number is properly set up, it is analyzed by the Valdec subprogram, which turns this ASCII string of numbers into an ordinary ML two-byte integer.

If, however, we were sent to EVEXFLAG (from **1710**), it checks for something less than an alphabetic character (such as a (or a # symbol). When it locates the first alphabetic character, it stores it into the variable WORK and JSRs off to the subprogram Array where the stored labels will be looked through. Then it joins up again with the numeric expressions by going to L340 for addressing mode evaluation.

How Is It Addressed?

This is the final job the assembler must perform—distinguishing between Immediate (LDA #15), Absolute (LDA 1500), Zero Page (LDA 15), Indirect Y (LDA (15),Y), and the other addressing modes. Recall that we've already eliminated nearly half the possibilities by previously handling type 0, the self-contained, implied ones like CLC and INY. What's left is to check for # and (symbols and to see how big the argument is. That tells us if our argument (the expression) calls for Zero Page addressing or not.

First off, LADS checks for the # character (**2130**) and, finding one, goes to the IMMED routine to handle Immediate addressing. Next it looks for the (character. Finding one of those, it goes off to the INDIR routine to deal with Indirect addressing.

Failing to find either of these symbols, it loads in the type variable, TP, and looks to see if it's an 8. All the B instructions, the branches like BNE and BCC, are grouped together as type 8. Finding a type 8, LADS goes to the REL subroutine to handle Relative addressing.

From here (line 2220) to the end of Eval, there will, from time to time, be adjustments made to the OP variable which are neither easy to explain nor easy to immediately understand. They're based on the logic of the interrelationships between the various addressing modes. For example, if we've reached this point (**2220**) without branching to one of the routines like IMMED, INDIR, or REL, we now need to add 8 to the opcode value. Why? It just works that way. If you're truly interested, study the table of opcodes and you'll begin to notice certain similarities between the opcode for LDA absolute and INC absolute, etc. It's not necessary to work all this out. For a detailed discussion of the logic of these adjustments to OP, see the explanation of the Tables subprogram in Chapter 9.

At any rate, INDIR looks at the character of the argument in BUFFER and sees if it's a) symbol. If not, and it's type 1, we add 16 to OP. If we have a type 6, we know we've got an indirect JMP, so we go there. Otherwise, we go to TWOS, where two-byte addressing modes, like LDA (15),Y, are handled.

JIMMED (**2420**) is one of those springboards to handle a BRANCH TOO FAR for an unassisted B instruction with its 127-byte reach.

The Hardest Part of LADS

REL handles the B group. This was the hardest part of LADS for me to write. For some reason, I kept hoping for a simple way to test and translate forward and backward branches. No simple way presented itself. There may be a more clever solution than the one you'll see described below, but I couldn't find it and had to go on.

REL first checks PASS. On pass 1, we simply go directly to TWOS. On pass 2, though, we look at RESULT. RESULT is a two-byte variable which holds the integer form of all arguments—labels, hex, or straight decimal. They're all left in RESULT by the various subprograms, Array, Indisk, and Valdec, which translate labels, hex ASCII, and decimal ASCII. These three possible original forms of the arguments are translated into two-byte integers that can be POKEd into memory or saved on disk as parts of an ML program.

If we're on pass 2, we look at RESULT and now calculate the correct argument for a branch instruction. It requires that LADS first determine whether we're branching backward or forward in memory. It does this by subtracting SA (the Program

Counter, the current address, the address of the B instruction to which its argument will be *relative*). It subtracts SA from RESULT, the argument of the B instruction:

```
100 1000 A0 00        START     LDY #0
110 1002 C8           LOOP      INY
120 1003 F0 03                  BEQ END
130 1005 4C 02 10               JMP LOOP
140 1008 60           END       RTS
```

The target, END, of the BEQ above is address 1008. The location of the PC at the BEQ is 1003. MREL (**2470**) first subtracts the PC in variable SA from the target's address. Remember that RESULT holds the correct integer after the Array subprogram looked through LADS' array and found the label END. So 1008 minus 1003 gives 5.

BPL and BMI

BCS tests the result of the subtraction—the carry is still set if the target is higher than SA and, consequently, we've got a branch forward. We BCS FOR. Otherwise, it's an attempt to branch backward in memory, and we test the high-byte result of the subtraction (the number in the accumulator) against $FF. That high byte must equal $FF, or we've branched too far and we go to the error-message printout routine (**2570**). Then we check the low-byte result of the subtraction (which was pushed on the stack temporarily in line 2500) to see if it's a correct value. The PLA (**2580**) will set the N flag in the Status Register if the number is greater than 127. We want it to be, since this is a backward branch. If this flag is not set, we BPL to the error message. Otherwise, we jump to the concluding routine, setting up a correct branch.

The FOR routine handles forward branches in a similar way, going to the error routine if the high byte is not zero (**2610**) or if the low byte has the seventh bit set (proving it's greater than 127, an incorrect forward branch).

Let's pause for a minute to see what BPL and BMI do for us in this test. In binary, $80 looks like this: 10000000. We don't care about the bits in the positions where the zeros are. We're only interested in the leftmost bit, the so-called seventh bit. Note, too, that PLA affects the N and Z flags in the Status Register.

After a PLA of 10000000, BPL would not branch anywhere, but BMI would. It would mean that the seventh bit is set, the "minus sign" in signed arithmetic was found. The *sign* in signed arithmetic is held in the seventh bit. 1XXXXXXX would signify a negative number, 0XXXXXXX a positive number. (There's a connection here with the fact that forward branch arguments can range from $00 to $7F, and backward branches from $FF down to $80.)

Now some people will point out that there are *eight* bits in a byte, and we keep referring to the seventh bit when we're talking about the eighth. Recall that, in computing, much counting begins with the zeroth bit. A byte can hold only the numbers 0–255. The lowest number it can hold is a zero. But that still means that there are 256 possibilities, 256 possible states for a byte: 1–255 plus 0.

Signed Arithmetic Branching

If all this seems an unnecessary detour into messy detail, consider how Relative addressing uses signed arithmetic to calculate where it should branch. When the 6502 chip comes upon one of the B branch instructions like BNE, it looks at the argument in a unique way. If the number is higher than 127, it knows it must go backward. If lower or equal, it must go forward. That's why you cannot branch further than 128 backward or 127 forward. The argument can't use the entire byte to hold a number—the seventh bit must be reserved to hold the plus or minus sign. Remember, if the seventh bit is set, it means minus. If clear, it means plus. BPL (Branch if PLus) is triggered when the seventh bit is clear. BMI responds to a set (1) seventh bit.

Take a look at the assembly in the example above. Line 120 shows that BEQ END became the opcode F0 and the argument is 03. 03 will take us to END because all branches are calculated from *the address of the mnemonic following the branch instruction*. Count three from address 1005. You hit END.

A branch backward, too, counts backward from the address of the mnemonic *following* the B instruction. All branches count from their own PC location *plus* 2. Look at a branch backward:

```
40 1000 A0 00     START    LDY #0
50 1002 C8        LOOP     INY
60 1003 D0 FD              BNE LOOP
70 1005 60        END      RTS
```

Here line 60 is branch backward, but the argument, $FD, is pretty strange. $FD looks like this in binary: 11111101. So the seventh bit is set signifying minus, a backward branch. $FD is 253 decimal. $FF would be −1, $FE would be −2, and $FD is −3. From address 1005, −3 lands us at 1002, LOOP, where we want to land. Luckily, we needn't perform these calculations. LADS will handle all branch arguments. But you might want to use BPL/BMI branches as well as signed arithmetic in your ML programming. It's sometimes worth knowing the details of how these things are handled by the microprocessor.

One final adjustment needs to be made before LADS can POKE in the correct argument for branches. This adjustment takes place at RELM, where both forward and backward branches end up, unless they were found to be out of range.

After the low byte of SA was subtracted from the low byte of RESULT (**2500**), we pushed it onto the stack with PHA. That's sometimes a convenient place to stuff something you want to set aside for a minute while you perform other calculations. You could STA A or STA TEMP or put it in other temporary holding variables, but PHA is safe *as long as you remember to PLA* to leave the stack clean. You don't want to keep PHAing, or your program will soon fill up the stack, resulting in an OVERFLOW error and a machine-wide collapse. The 6502 chip won't ignite, the CRT screen won't melt, but the program will grind to a halt.

When we have a BRANCH OUT OF RANGE error we are going to go down to the DOBERR routine at line 5800, but we do need to PLA in lines 2560 and 2620 to keep the stack clean.

If there is no error, we've saved the result of the subtraction of the low bytes (it sits in the low byte of the RESULT variable). That's the number we really care about anyway. A single byte is all that can be used as a branch argument.

To make it a correct branch argument, we've got to subtract 2 from it. This, you recall, is because all branches are calculated from the address of the mnemonic which comes *just after* the branch instruction. Counting starts from the B instruction's address, plus two. Subtracting two will fix this up for branches in either direction.

Further Evaluation

We've seen how LADS calculates the branch addresses. At this point in the source code, we come upon a continuation of evaluations of other addressing modes. EVM05 (**2740**) gets the size of the argument in order to enable us to look at the character second from the end: LDA (ZERO),Y has a comma in this second-from-the-end position. INX NAME does not. By now, the variety of possible addressing modes has been somewhat narrowed.

If we did find a comma in that second-from-last position, that means the label ends in ,X or ,Y and we go to XYTYPE to deal with it. Otherwise, we check to see if it's a JMP (opcode 76). MEV eliminates two other possible modes, both Zero Page, sending LADS to the TWOS, two-byte, line-ending events.

We're headed for TWOS by now in any case, but we need to once again adjust the value of the opcode in OP if the type in TP isn't 6 or 4.

TWOS, like TP1 (for one-byte-long instructions) and THREES, is where LADS goes after an addressing mode has been determined. The opcode has been correctly adjusted and waits in OP. The argument waits in RESULT. TP1, TWOS, and THREES are quite similar. TP1 doesn't have an argument, so it just JSRs to a subroutine within the subprogram Printops. There, the bytes are POKEd into memory or to disk and PRINTed to screen or printer. Then LADS JMPs to INLINE to prepare for the next line of code.

TWOS (**2970**) and THREES (**3400**) also JSR to that same subroutine in Printops (which POKEs, SAVEs, or PRINTs an opcode), and then TWOS and THREES JSR to PRINT2 or PRINT3 as appropriate to store or print the byte or bytes of the argument.

Immediate addressing (LDA #12) is a variation of TWOS, but it first must make one of those adjustments to the value of the opcode before JUMPing to TWOs (see line **950**).

THREES also requires some opcode adjustments before storing or printing its bytes; PREPTHREES (**3240–3390**) accomplishes that.

The JUMP subroutine (**3010**) handles the mnemonic JMP. It's a special case because it can have a strange addressing mode called Indirect Jump. JUMP tests for this and makes the necessary adjustment to the opcode if it finds the ASCII code for a parenthesis, indicating an Indirect Jump, for example JMP ($5000).

IMMED handles the # type, Immediate addressing. It first looks to see if the #" pseudo-op is in effect (**3100**) and, if so, stores the argument directly from the buffer. Then IMMED adjusts the base opcode (in the OP variable) if necessary, and behaves like any other two-byte addressing mode, jumping to TWOS.

Preparations for a New Line

We come now to the cleanup routine, INLINE (**3440**). Its primary job is to handle the correct formatting of the printout of the source code. By the time LADS gets to INLINE, it's already printed a line's number, the address of the PC (the location of the code), and the object code bytes themselves:

```
line #   /addr  /bytes of object code
40        1000   A0 00
```

However, there are still three items to print: an address label (if any), the source code, and remarks (if any). To make listings easy to read, address labels should be set off by themselves, and source code should line up vertically on a printed page or screen:

```
line #   /addr  /bytes / addr label  /source   / comments
40        1000   A0 00   START        LDY #0   ; begin here (entry)
```

Since each column should line up correctly, we're going to need to construct the ML equivalent of BASIC's TAB function. Those first three items—line number, address, and object code bytes—can take care of themselves. But any address labels must always be in the same position on a line. And since there can be one, two, or three object code bytes, the address labels wouldn't line up if we just printed a couple of spaces after the final object byte.

TAB

The first thing INLINE does is to check if we're on the first pass. Nothing gets printed out on pass 1, so we jump over the entire INLINE routine. If it's pass 2, we look to see if the screen flag, SFLAG, is up (**3470**). If it isn't, we again jump past INLINE.

Then the LOCFLAG is checked. It is up when there is a PC address label (like the label START in the example above). If it's up, we use something from BASIC: the cursor position byte. We've been using BASIC's PRINT routine all along. One of the advantages of this is that PRINT keeps a record in zero page of the current screen position; we could just LDA #20:STA CURPOS, and the next printout would be at position 20.

Tab to Printer

Things are more complicated, though, since LADS has an option to print listings to a printer as well as to the screen. We cannot use the same technique with a printer.

To find out how many blanks to print to the printer, it's necessary to subtract the CURPOS value from 20. Assume that we've printed 14 characters so far: 20 − 14 = 6. We use this result in a loop to print blanks to the printer (**3660**) to cause a simulated TAB.

Following the TAB, we're set to print an address label which is still waiting for us up in the buffer FILEN. As usual, we set TEMP to point to the message we want printed, and JSR PRNTMESS, thereby printing whatever is in FILEN, delimited by 0.

Source Code Printout

It's time to move over to the thirtieth position (on screen or printer) to the place where the source code is printed. This is handled basically the same way as the TAB 20 above. The main difference is the BEQ and BMI checks (**3920**) to take care of extra long labels. In most cases, your labels will be less than ten characters long, but LADS allows labels to be any length. How will we balance the need for neat, vertically aligned printouts against the option of labels of any length? How can labels which potentially range in length from 1 to 200 characters be formatted?

Since address labels always start in the twentieth position, and source code always begins in the thirtieth position, we've allowed ten spaces for address labels during printout. Onscreen, an address label 12 characters long would be *truncated*: STARTLINEHERE would be printed as STARTLINEH. But on the printer, the entire label would be printed and simply push the source code printout over. You can adjust any of these formatting options rather easily if they don't suit your needs. If you want to truncate address labels to five rather than ten character lengths on screen, just change LDA #30 to LDA #25 (**3830**).

In INLINE, we've done some output switching between screen and printer. We've called upon routines like CLRCHN, CHKOUT, and CHKIN. The protocol for using these routines is discussed in Chapter 5, the chapter on peripheral communications.

PRMMFIN (**4000**) prints the characters in the buffer LABEL. That will be the source code. Then, LADS checks to see if there was a < or > pseudo-op in this line. If so, it tags one of these symbols onto the end of the source code label. If your source code looks like this: LDA #>STARTLINE, the printout will be LDA #STARTLINE>. This will help to call attention to this special pseudo-op addressing mode. The < and > symbols are not buried within the label.

The underlying reason for doing things this way, however, is not its visual appeal. It's easier and faster for LADS to analyze #STARTLINE than to analyze #>STARTLINE. During the analysis phase, LADS pulls out the < or > and raises BYTLFAG to show that the pseudo-op was originally a part of the label. Then it can assemble the label the same way it would assemble any other label.

The final job to be performed by INLINE is to check BABFLAG to see if there is a REMark, a comment, to print out (**4100**). The Indisk subprogam sends any comments to the buffer called BABUF to keep them safely out of the way. BABUF is the same buffer that BASIC uses for input. If there is a comment, we print a semicolon (**4130**), point TEMP to BABUF (**4160**), and PRNTMESS.

Then a carriage return is printed and we check to see if this was the final line of the source code. If ENDFLAG is set, we go to the assembly shutdown routine, FINI. If not, we pop back up to where we first started this line, STARTLINE, and pull in the next line of source code.

FINI: Which Pass?

As a two-pass assembler, LADS, of course, goes through the entire source code twice. When we get to FINI, we need to check which pass we're on. If it's pass 1, we INC PASS (from its zero condition, thereby setting it). After this INC, the next time we reach the end of the source code and come to FINI, we'll be sent to FIN, the shutdown routine.

But assume we've just finished pass 1 at this point. What we must do is reset the PC, the Program Counter. Back at the beginning, we saved the starting address in TA. SA has been LADS' PC variable, counting up and always keeping track of the current address during each event. Now it's time to reset SA by putting TA in it. Then we close the source code file on disk and promptly open it up again. This has the effect of resetting the disk's PC to point to the first byte in the disk file. Now we're ready to read in the source code all over again. We're ready to start the second pass.

We jump back up, just below START, to SMORE and read in, once again, the first line of the entire source code.

If we've already completed pass 2, however, we don't want to restart source code examination—everything's already accomplished, POKEd and PRINTed and SAVEd to disk as the case may be. We want to gracefully exit the assembler. FIN (**4390**) does this. It closes down any read or write files on disk, closes down communication to a printer, and jumps to BASIC mode. Now would be the time to try the object code program, to make some adjustments to your source code if you want, and then SYS back into LADS for another assembly.

Each computer has a "side entrance," a warm start into its BASIC. This entrance doesn't wipe out what's in RAM memory, doesn't blank out the screen. It's here that the LADS goes to move gently back into BASIC mode. The address of TOBASIC for each computer is defined in the subprogam Defs.

Evaluating ,X and ,Y

Although FINI is the logical end of the evaluation process, it's not the physical end of the Eval subprogram. Just below FINI is XYTYPE where such addressing modes as LDA $5000,Y are analyzed.

They too require some opcode adjustments before going to TWOS or THREES for printing and POKEing. We JMP to XYTYPE after having found a comma in a source code line like:

LDA SCREEN,X

and so the Y Register is pointing to the character just beyond the comma when we arrive at XYTYPE. All we need to do is load BUFFER,Y to check if the character following the comma is an X or a Y. If it's an X, we jump down to L720 which handles X type modes.

Otherwise, we're dealing with something involving a Y addressing mode. It might be this:

LDA (15),Y

so we have to check for the right parenthesis. We DEY DEY to move over to just in front of the comma and see if there's a) symbol. If not, we've got a Zero Page Y addressing mode like LDX 10,Y or STX 10,Y. LDX and STX are the only two mnemonics which can use Zero Page Y addressing. They're rare. It's quite likely you haven't ever used them; it's possible that you haven't ever heard of them. But LADS must check for them just in case. LADS goes to ZEROY if there was no) symbol.

LADS is likely to find the), however, because Indirect Y addressing is a mode which is both common and useful. Encountering this mode, LADS goes to INDIR to process the Indirect addressing mode.

ZEROY (**4660**) is a somewhat misleading name, for it also handles the popular mode, Absolute Y: LDA SCREEN,Y. This addressing mode is not Zero Page. To find out whether it's dealing with the Zero Page Y, LADS checks the high byte of RESULT, the argument. If the high byte contains nothing, it must be zero page, and we process the opcode as such. If the high byte does contain something, the argument is thus larger than 255 and the opcode cannot use a Zero Page addressing mode. Again, the opcode is adjusted depending on the type (TP).

The routine at L700 (**4950**) prints out an error message because LADS was unable to calculate a correct addressing mode and the source code must contain a syntax error.

The concluding adjustments to the opcode take place between L720 and L809 (**5040–5450**). You might notice several JSRs to P in this section. P (**5520**) is a short subroutine which was used in debugging LADS, but was left in because you might want to use it when fixing up your own programs.

How P Works

P prints the current PC on screen, but doesn't destroy what's in the A, Y, or X Registers. Saving A, Y, and X is straightforward enough (**5520**), but where is the PC?

Whenever you JSR, the return address is pushed onto the stack. We can pull it off the stack with PLA, transferring its two bytes (one to the X Register and one to the Accumulator), and then push it back on with PHA. That leaves the stack ready to RTS correctly, but a copy of this RTS address is now in the registers as well, OUTNUM is a BASIC routine which normally prints line numbers during BASIC's LIST. But it will print any integer number if the low byte is in X and the high byte is in A. (See Atari notes for Atari's OUTNUM.)

Character $BA on Commodore machines is a check graphics symbol (✓), and it's a convenient way to show that what follows is not part of a normal LADS printout. You could use any other symbol to highlight the special nature of the number being printed by P. What's important is that you are alerted to the fact that somewhere within your ML program, you did JSR to P. And the number that P prints will be the address of that JSR.

How is P useful? An ML program is like a rocket. It's so fast that you need to send up balloons now and then just to mark its passage from subroutine to subroutine. When you're not getting what you expect (and that's often in large, interacting ML programs), you can put JSR P into various parts of the program. Then, as the program zips along, you'll be able to see what's happening and in what order it's happening.

P is like setting BRK into the code or putting STOP into a BASIC program. The difference is that P just gives you a simple location report and lets the program continue, uninterrupted. If you wanted more information, you could expand P to print the registers at the same time. With that, you'd be on your way toward constructing the single-step debugging feature available in some monitor programs.

CLEANLAB (**5720**) is janitorial. It wipes the main buffers clean. It puts 80 zeros into LADS' main input buffer starting at LABEL (see Chapter 9, where the Tables are described). We don't want any remnants of the previous line left over to confuse things.

Finally, DOBERR is the error message printout routine for branches out of range. It rings the bell (ERRING), prints the offending line number, then points TEMPS to its message (stored with the other messages in the Tables subprogram), and jumps to TWOS so that the Program Counter will still be correctly increased by two.

Now we've seen the innards of Eval, the main evaluation engine, the heart of the LADS assembler. It's time to turn our attention to the data base managers Equate and Array. They build and search the array of labels.

Program 3-1. Eval

```
10 ; "EVAL"  MAIN EVALUATION ROUTINE (SIMPLE ASSEMBLER)
20 ;---------------------------------------------------------------
30 START LDA #0
40 LDY #48
50 STRTLP STA OP,Y;      -- LOOP TO CLEAR FLAGS --
60 DEY
70 BNE STRTLP; --------
80 LDA #<START; STORE BOTTOM OF LADS INTO TOP OF ARRAY/MEMORY.  PROTECT IT.
90 STA MEMTOP
100 STA BMEMTOP
110 STA ARRAYTOP
120 LDA #>START
130 STA MEMTOP+1
140 STA BMEMTOP+1
150 STA ARRAYTOP+1;--------
160 LDA #1;              -- SET DEFAULTS --
170 ; HERE YOU CAN SET ANY ADDITIONAL DEFAULTS YOU WISH
180 STA HXFLAG; TURN ON HEX LISTING FLAG
190 STM0 LDA SCREEN,Y;             -- GET SOURCE FILE NAME --
200 CMP #32
210 BEQ STM1; CHECK FOR ANOTHER BLANK
220 BCS STM3
230 CLC
240 ADC #64; ADJUST FOR LOW ASCII CHARACTERS
250 STM3 STA FILEN,Y; STORE CHARACTER IN FILENAME BUFFER
260 INY
270 JMP STM0; GET ANOTHER CHARACTER
280 STM1 STA FILEN,Y; CHECK FOR 2ND BLANK
290 INY
```

```
300 LDA SCREEN,Y
310 CMP #32; IF NO 2ND BLANK SPACE
320 BNE STM0;             THEN GO BACK FOR MORE NAME (MIGHT BE 2 WORDS)
330 DEY
340 STY FNAMELEN;  STORE FILENAME LENGTH
350 JSR OPEN1; OPEN READ FILE (SOURCE CODE FILE ON DISK)
360 ;------------------  RE-ENTRY POINT FOR PASS 2 -------
370 SMORE JSR GETSA; POINT DISKFILE TO 1ST CHARACTER IN SOURCE CODE
380 LDA #0
390 STA ENDFLAG; SET LADS-IS-OVER FLAG TO DOWN
400 JSR INDISK; GET A SINGLE LINE OF SOURCE CODE
410 LDA PASS; IF 2ND PASS
420 BNE STARTLINE;          THEN JUMP OVER PRINTING OF LADS NAME
430 JSR PRNTCR; PRINT CARRIAGE RETURN
440 LDA #230; PRINT BLOCK GRAPHICS SYMBOL
450 JSR PRINT
460 LDA #76;         L
470 JSR PRINT
480 LDA #65;         A
490 JSR PRINT
500 LDA #68;         D
510 JSR PRINT
520 LDA #83;         S
530 JSR PRINT
540 JSR PRNTCR; ANOTHER CARRIAGE RETURN
550 CKHEX LDA HEXFLAG; IF START ADDRESS NUMBER IS HEX, IT'S ALREADY TRANSLATED
560 BNE STAR1
570 LDA #<LABEL; IN THE LABEL BUFFER IS SOMETHING LIKE: *= 864
580 STA TEMP; PUT THE ADDRESS OF THE BUFFER INTO THE POINTER CALLED TEMP
590 LDA #>LABEL
600 STA TEMP+1
```

```
610 JSR VALDEC; TURN ASCII NUMBER INTO A TWO-BYTE INTEGER IN "RESULT"
620 STAR1 LDA RESULT;     -- STORE OBJECT CODE'S STARTING ADDRESS IN SA,TA --
630 STA SA
640 STA TA
650 LDA RESULT+1
660 STA SA+1
670 STA TA+1
680 ;----------------   ENTRY POINT FOR EACH NEW LINE OF SOURCE CODE  ----
690 STARTLINE JSR STOPKEY:LDA ENDFLAG:BEQ EVIND:JMP FINI; END LADS ASSEMBLY IF
700 ; EITHER THE STOP (BREAK) KEY IS PRESSED OR IF THE ENDFLAG IS UP.
710 ;
720 EVIND JSR INDISK; OTHERWISE GO TO PULL IN A LINE FROM SOURCE CODE
730 LDA #0
740 STA EXPRESSF; SET DOWN THE FLAG THAT SIGNALS A LABEL ARGUMENT LIKE LDA P
750 STA BUFLAG; SET DOWN THE FLAG THAT SIGNALS # OR ( DURING ARRAY CHECK.
760 LDY PASS;ON PASS 1, WE DON'T PRINT LINE NUMBERS, ADDR. OR ANYTHING ELSE
770 BNE MOREEV
780 JMP MOE4
790 MOREEV STY LOCFLAG; ZERO ADDRESS-TYPE LABEL FLAG (LIKE: LABEL INY)
800 ;                   THIS IS FOR THE INLINE SUBROUTINE BELOW.
810 LDA SFLAG; SHOULD WE PRINT TO THE SCREEN
820 BEQ MX; IF NOT, SKIP THIS PART
830 JSR PRNTLINE; PRINT LINE NUMBER
840 JSR PRNTSPACE; PRINT SPACE
850 JSR PRNTSA; PRINT PC (PROGRAM COUNTER)."SA" IS THE VARIABLE.
860 JSR PRNTSPACE
870 MX LDA PLUSFLAG; DO WE HAVE A + PSEUDO OP
880 BEQ MOE4; IF NOT SKIP
890 JSR MATH; IF SO, HANDLE IT IN SUBPROGRAM "MATH"
900 MOE4 JMP FINDMN; LOOK UP MNEMONIC (OR, NOT FINDING ONE, IT'S A LABEL)
910 ; --------  EVALUATE ARGUMENT
```

```
920 EVAR LDA TP
930 BEQ TP1JMP; CHECK TYPE, IF Ø, NO ARGUMENT
940 CMP #3; IF NOT TYPE 3, THEN CONTINUE EVALUATION
950 BNE EVGO
960 LDA #1; OTHERWISE, REPLACE 3 WITH 1 IN TP (TYPE)
970 STA TP
980 LDA LABEL+3; IS THERE SOMETHING (NOT A ZERO) IN 4TH POSITION
990 BNE EVGO; EVGO = ARGUMENT (IF NOT, THERE'S NO ARGUMENT,IT'S IMPLIED
1000 LDA #8; OTHERWISE, RAISE OP (OPCODE) BY 8
1010 CLC
1020 ADC OP
1030 STA OP
1040 TP1JMP JMP TP1; AND JUMP TO TYPE 1 (SINGLE BYTE TYPES)
1050 ;------------------
1060 EQLABEL LDA PASS; MOE4 FOUND IT TO BE A LABEL, NOT A MNEMONIC
1070 BEQ EQLAB1; ON PASS 1 WE DON'T CARE WHICH KIND OF LABEL IT IS SO WE
1080 LDY #255; GO DOWN AND STORE IT IN THE ARRAY (VIA EQLAB1)
1090 EVX1 INY; BUT ON PASS 2, WE NEED TO DECIDE IF IT'S A PC ADDRESS TYPE
1100 LDA LABEL,Y; LABEL (LIKE: LABEL INY) OR AN EQUATE TYPE (LABEL = 15)
1110 BEQ GONOAR; SO IN THIS LOOP WE LOOK FOR A BLANK WHILE STORING THE
1120 STA FILEN,Y; LABEL NAME IN THE "FILEN" BUFFER. IF WE FIND A Ø, IT'S
1130 CMP #32; A NAKED LABEL (NO ARGUMENT TO IT) WHICH CAUSES US TO PRINT
1140 BNE EVX1;OUT THAT ERROR MESSAGE (AT NOAR, IN EQUATE).OTHERWISE, WE FIND A
1150 INY; BLANK AND FALL THROUGH TO THIS LINE.
1160 LDA LABEL,Y; WE RAISE Y BY 1 AND CHECK FOR AN = SIGN.
1170 CMP #$3D
1180 BNE NOTEQ; IF NOT, IT'S A PC ADDRESS TYPE (SO SET LOCFLAG)
1190 JMP INLINE; IF SO,WAS = TYPE SO IGNORE IT (ON PASS 2) ----------
1200 NOTEQ LDX #Ø
1210 STX LOCFLAG;(SHOWS PRINTOUT TO DO THIS TYPE OF LABEL ON SCREEN/PRINTER)
1220 TXA; PUT A ZERO IN AT THE END OF THE LABEL NAME (AS A DELIMITER)
```

```
1230 STA FILEN,Y; NOW WE HAVE TO MOVE THE ARGUMENT PORTION OF THIS LINE
1240 EVX5 LDA LABEL,Y; OVER TO THE START OF THE "LABEL" BUFFER FOR FURTHER
1250 BEQ EVX4; ANALYSIS (Ø DELIMITER HERE)
1260 STA LABEL,X; WE CAN IGNORE THE PC LABEL (THIS IS PASS 2), BUT WE
1270 INX; NEED TO EVALUATE THE REST OF THE LINE FOLLOWING THAT LABEL.
1280 INY
1290 JMP EVX5;-------------------
1300 EVX4 STA LABEL,X
1310 JMP MOE4; JUMP TO CONTINUE EVALUATION
1320 GONOAR JSR NOAR; PRINT NO ARGUMENT MESSAGE (A SPRINGBOARD);----------
1330 EQLAB1 JSR EQUATE; PUT LABEL AND IT'S VALUE INTO THE ARRAY (PASS 1)
1340 JMP MOE4; CONTINUE EVALUATION
1350 ;------------------ TRANSLATE ARGUMENT LABELS INTO NUMBERS
1360 EVEXLAB LDA BUFFER; IS THIS 1ST CHARACTER ALPHABETIC (>64)
1370 CMP #64
1380 BCS EVE1; IF SO, GO DOWN TO FIND ITS VALUE.
1390 LDA BUFFER+1; IF NOT, IT MUST HAVE BEEN A ( OR # SYMBOL
1400 INC BUFLAG; TO TELL ARRAY THAT ( OR # WAS FOUND (AND TO IGNORE THEM)
1410 EVE1 EOR #$8Ø; SET 7TH BIT IN 1ST CHAR. (TO MATCH ARRAY STORAGE METHOD)
1420 STA WORK; SAVE IT HERE TEMPORARILY TO COMPARE WITH ARRAY WORDS
1430 JSR ARRAY;  EVAL. EXPRESSION LABEL, SHIFTED 1ST CHAR.
1440 JMP L34Ø; THEN CONTINUE ON WITH EVALUATION (AFTER VALUE IS IN "RESULT")
1450 ;--------------- IS ARGUMENT NUMERIC OR A LABEL
1460 EVGO LDY #Ø
1470 STY EXPRESSF; TURN OFF THE "IT'S A LABEL" FLAG
1472 ;------------------- SEE CHAPTER 11 FOR DESCRIPTION OF THIS ERROR TRAP
1473 ;                                (TRAP FOR NAKED MNEMONICS ERROR)
1474 LDA LABEL+3:CMP #32:BEQ GVEG:JMP L7ØØ; (TEST FOR "INC:" TYPE ERROR)
1480 GVEG LDA LABEL+4,Y; CHECK 5TH CHAR. (LDA NAME OR LDA 25) (THE "N" OR "2")
1490 CMP #65; IF LESS THAN 65 (ASCII FOR "A") THEN IT'S A NUMBER
1500 BCC EVMO2A
```

```
1510 INC EXPRESSF; >65 = ALPHABETIC ARG (LABEL) SO RAISE THIS FLAG
1520 EVMO2A STA BUFFER,Y; STORE 1ST CHAR. OF ARGUMENT IN "BUFFER" BUFFER
1530 INY
1540 LDA LABEL+4,Y; LOOK AT 2ND CHAR. IN THE ARGUMENT
1550 BEQ EVMO3; IF ZERO, WE'RE AT THE END SO MOVE ON
1560 STA BUFFER,Y; OTHERWISE, STORE 2ND CHAR.
1570 CMP #65; IF LOWER THAN 65, DON'T RAISE LABEL-ARGUMENT FLAG
1580 BCC EVMO2
1590 INC EXPRESSF; IF HIGHER, DO RAISE IT
1600 EVMO2 INY; NOW MOVE REST OF ARGUMENT UP TO "BUFFER" BUFFER
1610 LDA LABEL+4,Y; LOOP TO MOVE THE ARGUMENT INTO THE BUFFER
1620 BEQ EVMO3; EVMO3 TAKES OVER AFTER END OF ARGUMENT IS REACHED
1630 STA BUFFER,Y
1640 JMP EVMO2; RETURN FOR MORE ARGUMENT CHARACTERS.
1650 ;---------------
1660 EVMO3 DEY
1670 STY ARGSIZE; REMEMBER NUMBER OF CHARACTERS IN ARGUMENT
1680 LDA HEXFLAG; IF IT'S HEX, INDISK SUBPROGRAM ALREADY TRANSLATED IT FOR US
1690 BNE L340; SO GO ON TO EVALUATE ADDRESS MODE.
1700 LDA EXPRESSF; IF IT'S A LABEL (NOT A NUMBER) THEN GO TO THE ROUTINE
1710 BNE EVEXLAB; WHICH EVALUATES EXPRESSION (ARGUMENT) LABELS, "EVEXLAB"
1720 ; ----------  CALCULATE ARGUMENT'S VALUE (IF IT'S A DECIMAL NUMBER)
1730 LDA #<BUFFER; MAKE "TEMP" POINTER POINT TO "BUFFER"
1740 STA TEMP
1750 LDA #>BUFFER
1760 STA TEMP+1
1770 LDY #0
1780 LDA BUFFER; IS 1ST CHARACTER HIGHER THAN 48 (ASCII FOR THE NUMBER ZERO)
1790 CMP #48
1800 BCS MCAL; IF SO, SKIP THIS PART
1810 CLC; IF NOT, THE 1ST CHARACTER MUST BE # OR ( ..... SO WE NEED TO
```

```
1820 INC TEMP; MAKE "TEMP" POINT 1 CHARACTER HIGHER IN "BUFFER" TO
1830 BCC MCAL; AVOID HAVING THE ASCII TO INTEGER SUBROUTINE THINK THAT THE
1840 INC TEMP+1; NUMBER STARTS WITH A # OR ( --- THAT WOULD MESS THINGS UP.
1850 MCAL LDA (TEMP),Y; NOW LOOK FOR THE END OF THE NUMBER:  -----------
1860 BEQ MCAL1; IT COULD END WITH A Ø (DELIMITER) OR
1870 CMP #41; WITH A ) RIGHT PARENTHESIS OR
1880 BEQ MCAL1
1890 CMP #44; WITH A , COMMA (AS IN: 15,Y) OR
1900 BEQ MCAL1
1910 CMP #32; WITH BLANK SPACE (AS IN: #15  ;COMMENT)
1920 BEQ MCAL1
1930 INY; IF WE'VE NOT YET FOUND ONE OF THESE 4 THINGS, CONTINUE LOOKING
1940 JMP MCAL;---------------------------
1950 MCAL1 PHA; SAVE ACCUMULATOR
1960 TYA
1970 PHA;SAVE Y REGISTER(BY NOW, Y IS POINTING AT THE SPACE JUST AFTER THE #)
1980 LDA #Ø; PUT DELIMITER ZERO INTO BUFFER JUST FOLLOWING NUMBER.
1990 STA (TEMP),Y
2000 JSR VALDEC;GO TO THE ASCII-NUMBER-TO-INTEGER-NUMBER-IN-"RESULT" ROUTINE
2010 PLA; RESTORE THE A AND Y REGISTERS
2020 TAY
2030 PLA
2040 STA (TEMP),Y; RESTORE "," OR ")" TO THE BUFFER (FOR THE ADDR. ANALYSIS)
2045 ;
2050 ;---------  ANALYZE THE ARGUMENT TO DETERMINE ADDRESSING MODE ------
2055 ;
2060 ; (THIS ESSENTIALLY AMOUNTS TO MODIFYING THE ORIGINAL OPCODE TO
2070 ;  REFLECT THE CORRECT ADDRESSING MODE.  ADJUSTMENTS TO THE OPCODE "OP"
2080 ;  APPEAR RATHER FREQUENTLY FROM HERE ON.  THEIR LOGIC WILL NOT BE
2090 ;  COMMENTED.  ADDING 4,8,16, OR 24 TO AN "OP" IS BASED ON THE
2100 ;  RELATIONSHIPS WITHIN THE OPCODE TABLE (SEE CHAPTER 9 FOR EXPLANATION)
```

```
2110 ;
2120 L340 LDA BUFFER; 1ST CHAR. OF THE ARGUMENT (THE "#" IN LDA #15)
2130 CMP #35
2140 BEQ JIMMED; # SYMBOL FOUND (SO IMMEDIATE MODE).  BRANCH TO SPRINGBOARD
2150 CMP #40; IS IT A "(" LEFT PARENTHESIS.  IF SO, GO TO INDIRECT ADDR.
2160 BEQ INDIR
2170 LDA TP; IS IT A RELATIVE ADDR. MODE (LIKE BNE, BEQ).
2180 CMP #8
2190 BEQ REL; IF SO, GO TO WHERE THEY ARE HANDLED.
2200 CMP #3; ADD 8 TO OP AT THIS POINT IF IT'S A TYPE 3
2210 BNE EVMO5
2220 LDA #8
2230 CLC
2240 ADC OP
2250 STA OP
2260 JMP TP1; AND JUMP TO THE SINGLE BYTE TYPES (IMPLIED ADDRESSING)
2270 INDIR LDY ARGSIZE; HANDLE INDIRECT ADDRESSING------------------
2280 LDA BUFFER,Y; LOOK AT THE LAST CHARACTER IN THE ARGUMENT.
2290 CMP #41; IS IT A ")" RIGHT PARENTHESIS
2300 BEQ MINDIR; IF SO, HANDLE THAT TYPE.
2310 LDA TP
2320 CMP #1; IF TYPE 1, ADD 16 AT THIS POINT TO OPCODE
2330 BNE MINDIR
2340 LDA #16
2350 CLC
2360 ADC OP
2370 STA OP
2380 MINDIR LDA TP; TYPE 6 IS A JUMP INSTRUCTION
2390 CMP #6
2400 BEQ JJUMP; SO GO TO THE JUMP-HANDLING ROUTINE
2410 JMP TWOS; OTHERWISE, IT MUST BE A 2-BYTE TYPE SO PRINT/POKE IT.;-------
2420 JIMMED JMP IMMED; SPRINGBOARD TO IMMEDIATE MODE TYPES.
```

```
2430 ;-------------------------------- HANDLE RELATIVE ADDRESS (BNE) TYPES
2440 REL LDA PASS; ON PASS 1, DON'T BOTHER, JUST INCREASE PC BY 2
2450 BNE MREL
2460 JMP TWOS
2470 MREL SEC; ON PASS 2, SUBTRACT PC FROM ARGUMENT TO GET REL. BRANCH
2480 LDA RESULT
2490 SBC SA
2500 PHA; SAVE LOW BYTE ANSWER
2510 LDA RESULT+1
2520 SBC SA+1
2530 BCS FOR; IF ARGUMENT > CURRENT PC, THEN IT'S A BRANCH FORWARD
2540 CMP #$FF
2550 BEQ MPXS
2560 PLA
2570 JMP DOBERR
2580 MPXS PLA; OTHERWISE, CHECK FOR OUT OF RANGE BRANCH ATTEMPT
2590 BPL BERR; OUT OF RANGE (PRINT ERROR MESSAGE "BERR")
2600 JMP RELM; AND JUMP TO REL CONCLUSION ROUTINE
2610 FOR BEQ MPXS1; CHECK FORWARD BRANCH OUT OF RANGE
2620 PLA
2630 JMP DOBERR
2640 MPXS1 PLA
2650 BPL RELM; WITHIN RANGE-----------------------
2660 BERR JMP DOBERR; PRINT "BRANCH OUT OF RANGE" ERROR MESSAGE
2670 RELM SEC; FINISH UP REL. ADDR. TYPE --------------------
2680 SBC #2; CORRECT FOR THE FACT THAT BRANCHES ARE CALCULATED FROM THE
2690 STA RESULT; INSTRUCTION FOLLOWING THEM:  BNE LOOP:LDA 15 WOULD BE
2700 LDA #0;      CALCULATED FROM THE PC OF THE LDA 15
2710 STA RESULT+1
2720 JMP TWOS; NOW GO TO THE 2-BYTE PRINT/POKE (WITH CORRECT ARGUMENT)
2730 ;-------------------------------- CONTINUE ADDR. MODE ANALYSIS
```

```
2740 EVMO5 LDY ARGSIZE
2750 DEY
2760 LDA BUFFER,Y; LOOK AT LAST CHARACTER OF ARGUMENT
2770 CMP #44; IF IT'S NOT A COMMA, THEN THIS MUST BE A JUMP INSTRUCTION
2780 BNE JJUMP; SO GO TO THE JUMP-HANDLING ROUTINE
2790 INY
2800 JMP XYTYPE; OTHERWISE, IT MUST BE A ,X OR ,Y TYPE;---------------
2810 JJUMP LDA OP; HANDLE JMP MNEMONIC
2820 CMP #76; IF THE OPCODE ISN'T 76, IT'S NOT A JUMP
2830 BNE MEV; SO LOOK FOR SOMETHING ELSE
2840 JMP JUMP; NOW SPRINGBOARD TO THE JUMP-HANDLING ROUTINE.----------
2850 MEV LDA RESULT+1; IF HIGH BYTE OF RESULT ISN'T ZERO (ZERO PG. ADDR)
2860 BNE PREPTHREES; THEN GO TO THE 3-BYTE INSTRUCTIONS (LINE 400)
2870 LDA TP; OTHERWISE, IT'S ZERO PAGE MODE
2880 CMP #6; IF HIGHER THAN TYPE 6, IT'S AN ORDINARY 2-BYTE TYPE
2890 BCS TWOS; SO GO THERE.
2900 CMP #2; IF TYPE 2, ALSO GO THERE.
2910 BEQ TWOS
2920 LDA #4; OTHERWISE, ADD 4 TO OPCODE AND FALL THROUGH INTO TWO-BYTE TYPE
2930 CLC
2940 ADC OP
2950 STA OP
2960 ;-------------------------------  2 BYTE TYPES (LIKE LDA 12)
2970 TWOS JSR FORMAT; PRINT/POKE OPCODE
2980 JSR PRINT2; THEN PRINT/POKE ARGUMENT
2990 JMP INLINE; AND FINALLY PREPARE TO FETCH NEW LINE OF SOURCECODE (2000)
3000 ;-------------------------------  HANDLE JMP
3010 JUMP LDY ARGSIZE; IS IT JMP 1500 OR JMP (1500)
3020 LDA BUFFER,Y;  )  AT THE END PROVES IT'S AN INDIRECT JUMP SO
3030 CMP #41
3040 BNE JUMO
```

```
3050 LDA #108;          WE MUST CHANGE THE OPCODE FROM 76 TO 108
3060 STA OP
3070 JUMO JMP THREES; TREAT IT AS A NORMAL 3-BYTE INSTRUCTION
3080 ;-------------------------------- IMMEDIATE ADDRESSING (# TYPE)
3090 IMMED LDA BUFFER+1
3100 CMP #""; IS THIS A CHARACTER LOAD PSEUDO-OP LIKE: LDA #"A
3110 BNE IMMEDX
3120 LDA BUFFER+2; IF SO, PUT THE ASCII CHAR. INTO "RESULT" (ARGUMENT)
3130 STA RESULT
3140 IMMEDX LDA TP
3150 CMP #1
3160 BNE TWOS; IF IT'S TYPE 1, ADJUST OPCODE BY ADDING 8 TO IT.
3170 LDA #8
3180 CLC:ADC OP:STA OP
3190 JMP TWOS
3200 ;-------------------------------- 1 BYTE TYPES
3210 TP1 JSR FORMAT; JUST POKE OPCODE FOR THESE, THERE'S NO ARGUMENT
3220 JMP INLINE; (LINE 1000)
3230 ;-------------------------------- 3 BYTE TYPES
3240 PREPTHREES LDA TP; SEVERAL OPCODE ADJUSTMENTS (BASED ON TYPE)
3250 CMP #2
3260 BEQ PTT
3270 CMP #7; (LINE 430)
3280 BNE PT1
3290 PTT LDA OP
3300 CLC
3310 ADC #8
3320 STA OP
3330 JMP THREES
3340 PT1 CMP #6
3350 BCS THREES
```

```
336Ø LDA OP
337Ø CLC
338Ø ADC #12
339Ø STA OP
34ØØ THREES JSR FORMAT; PRINT/POKE OPCODE
341Ø JSR PRINT3; PRINT/POKE 2 BYTES OF THE ARGUMENT   (3ØØØ)
342Ø ;------------------------------------ PREPARE TO GET A NEW LINE
343Ø ;PRINT MAIN INPUT AND COMMENTS, THEN TO STARTLINE
344Ø INLINE LDA PASS; ON PASS 1, IGNORE THIS WHOLE PRINTOUT THING.
345Ø BNE NLOX1
346Ø JMP JST
347Ø NLOX1 LDA SFLAG; LIKEWISE, IF SCREENFLAG IS DOWN, IGNORE.
348Ø BNE NLOX
349Ø JMP JST
35ØØ NLOX LDA LOCFLAG; ANY PC ADDRESS LABEL TO PRINT
351Ø BNE PRMMX1; NO LOC TO PRINT (RVS FLAG USAGE, FOR SPEED)
352Ø LDA PRINTFLAG; PRINT TO PRINTER
353Ø BEQ PRMM
354Ø LDA #2Ø
355Ø SEC
356Ø SBC CURPOS; SUBTRACT CURRENT CURSOR POSITION
357Ø STA A; MOVE THE CURSOR TO 2ØTH COLUMN ON THE SCREEN
358Ø JSR CLRCHN; PREPARE PRINTER TO PRINT BLANKS
359Ø LDX #4
36ØØ JSR CHKOUT
361Ø LDY A
362Ø BPL PRXM1
363Ø LDY #2
364Ø JMP PRMLOP
365Ø PRXM1 LDA #32
366Ø PRMLOP JSR PRINT;---------------- PRINT BLANKS TO PRINTER
```

```
3670 DEY
3680 BNE PRMLOP; PRINT MORE BLANKS TO PRINTER;---------------
3690 JSR CLRCHN; RESTORE NORMAL I/O
3700 LDX #1
3710 JSR CHKIN
3720 PRMM LDA #20; PUT 20 INTO CURRENT SCREEN CURSOR POSITION
3730 STA CURPOS
3740 LDA #<FILEN; POINT "TEMP" TO PC ADDRESS LABEL FOR PRINTOUT
3750 STA TEMP
3760 LDA #>FILEN
3770 STA TEMP+1
3780 JSR PRNTMESS; PRINT LOCATION LABEL;----------
3790 PRMMX1 LDA #30; MOVE CURSOR TO 30TH COLUMN
3800 SEC
3810 SBC CURPOS
3820 STA X; SAVE OFFSET FROM CURRENT POSITION (30-POSITION) FOR PRINTER
3830 LDA #30
3840 STA CURPOS; SET SCREEN CURSOR POSITION TO 30
3850 LDA PRINTFLAG; DO WE NEED TO PRINT BLANKS TO THE PRINTER
3860 BEQ PRMMFIN
3870 JSR CLRCHN; ALERT PRINTER TO RECEIVE BLANKS
3880 LDX #4
3890 JSR CHKOUT
3900 LDY X
3910 BEQ PXMX; HANDLE NO BLANKS (IGNORE)
3920 BMI PXMX; HANDLE TOO MANY BLANKS (>127) (IGNORE)
3930 LDA #32
3940 PRMLOPX JSR PRINT; PRINT BLANKS TO PRINTER FOR FORMATTING--------
3950 DEY
3960 BNE PRMLOPX; PRINT MORE BLANKS----------
3970 PXMX JSR CLRCHN; RESTORE NORMAL I/O
```

```
3980 LDX #1
3990 JSR CHKIN;---------------------------------------------------
4000 PRMMFIN JSR PRNTINPUT; PRINT MAIN INPUT BUFFER (BULK OF SOURCE LINE)
4010 LDA BYTFLAG; IS THERE A < OR > PSEUDO-OP TO PRINT --------------
4020 BEQ PRXM; HANDLE < AND >
4030 CMP #1; 1 IN BYTFLAG MEANS <
4040 BNE MO5
4050 LDA #60
4060 JMP PRMO
4070 MO5 LDA #62; PRINT >
4080 PRMO JSR PRINT
4090 JSR PTP1; PRINT > OR <.  PTP1 IS TO PRINTER----------
4100 PRXM LDA BABFLAG; IS THERE ANY COMMENT TO PRINT (SOMETHING FOLLOWING ;)
4110 BEQ RETTX; IF NOT, SKIP THIS.
4120 JSR PRNTSPACE; PRINT A SPACE--------  PRINT COMMENTS FIELD ----------
4130 LDA #59; PRINT A SEMICOLON
4140 JSR PRINT
4150 LDA #<BABUF; POINT "TEMP" TO THE COMMENTS BUFFER "BABUF"
4160 STA TEMP
4170 LDA #>BABUF
4180 STA TEMP+1
4190 JSR PRNTMESS; PRINT WHAT'S IN THE COMMENTS BUFFER
4200 RETTX JSR PRNTCR; PRINT CARRIAGE RETURN
4210 LDA ENDFLAG; IF ENDFLAG IS UP, JUMP TO THE SHUTDOWN ROUTINE
4220 BNE FINI
4230 JST JMP STARTLINE; OTHERWISE GO BACK UP TO GET THE NEXT SOURCE LINE.
4240 ;------------------------------------- THE END OF A PASS (1 OR 2)
4250 FINI LDA PASS
4260 BNE FIN; IF IT'S PASS 2, SHUT EVERYTHING DOWN.
4270 INC PASS; OTHERWISE, CHANGE PASS 1 TO PASS TWO (IN THE FLAG)
4280 LDA TA; PUT THE ORIGINAL START ADDR. INTO THE PC PROGRAM COUNTER (SA)
```

```
4290 STA SA
4300 LDA TA+1
4310 STA SA+1
4320 JSR CLRCHN; RESTORE ORDINARY I/O CONDITIONS
4330 LDA #1
4340 JSR CLOSE; CLOSE INPUT FILE
4350 JSR OPEN1; OPEN INPUT FILE (POINT IT TO THE 1ST BYTE IN THE FILE)
4360 JMP SMORE; PASS 1 FINISHED, START PASS 2 (ENTRY POINT FOR PASS 2)------
4370 ;
4380 ;------------------ SHUT DOWN LADS OPERATIONS AND RETURN TO BASIC -----
4390 FIN JSR CLRCHN; RESTORE NORMAL I/O
4400 LDA #1
4410 JSR CLOSE; CLOSE SOURCE CODE INPUT FILE
4420 LDA #2
4430 JSR CLOSE; CLOSE OBJECT CODE OUTPUT FILE (IF ANY)
4440 LDA PRINTFLAG; IS THE PRINTER ACTIVE
4450 BEQ FINFIN; IF NOT, JUST RETURN TO BASIC
4460 JSR CLRCHN; OTHERWISE SHUT DOWN PRINTER, GRACEFULLY.
4470 LDX #4
4480 JSR CHKOUT
4490 LDA #13;        BY PRINTING A CARRIAGE RETURN
4500 JSR PRINT
4510 JSR CLRCHN
4520 LDA #4
4530 JSR CLOSE
4540 FINFIN JMP TOBASIC; RETURN TO BASIC
4550 ;
4560 ;-----------------------------------   ,X OR ,Y ADDRESSING TYPE
4570 XYTYPE LDA BUFFER,Y; LOOK AT LAST CHAR. IN ARGUMENT
4580 CMP #88; IS IT AN X
4590 BEQ L720
```

```
4600 DEY; OTHERWISE, LOOK AT THE 3RD CHAR. FROM END OF ARGUMENT
4610 DEY
4620 LDA BUFFER,Y; IS IT A  )  RIGHT PARENTHESIS
4630 CMP #41
4640 BNE ZEROY; IF NOT, IT'S NOT AN INDIRECT ADDR. MODE
4650 JMP INDIR; IF SO, IT IS AN INDIRECT ADDRESSING MODE
4660 ZEROY LDA RESULT+1; CHECK HIGH BYTE OF RESULT (ZERO PG. OR NOT)
4670 BNE L680; ZERO Y TYPE
4680 LDA TP; ADJUST OPCODE BASED ON TYPE
4690 CMP #2
4700 BEQ L730
4710 CMP #5
4720 BEQ L730
4730 CMP #1
4740 BEQ L760
4750 L680 LDA TP
4760 CMP #1
4770 BNE L690
4780 LDA OP
4790 CLC
4800 ADC #24
4810 STA OP
4820 JMP THREES
4830 L690 LDA TP
4840 CMP #5
4850 BEQ M6
4860 LDA #$31
4870 JSR P
4880 JMP L700
4890 M6 LDA OP
4900 CLC
```

```
4910 ADC #28
4920 STA OP
4930 JMP THREES
4940 ;----------- PRINT A SYNTAX ERROR MESSAGE -----------------
4950 L700 JSR ERRING; RING ERROR BELL AND TURN ON REVERSE CHARACTERS
4960 JSR PRNTLINE; PRINT LINE NUMBER
4970 LDA #<MERROR; POINT "TEMP" TO SYNTAX ERROR MESSAGE
4980 STA TEMP
4990 LDA #>MERROR
5000 STA TEMP+1
5010 JSR PRNTMESS:JSR PRNTCR; PRINT THE MESSAGE
5020 JMP INLINE; GO TO THE GET-THE-NEXT-LINE ROUTINE
5030 ;------------------------------ CONTINUE ANALYSIS OF ADDR. MODE
5040 L720 LDA RESULT+1; MAKE FURTHER ADJUSTMENTS TO OPCODE
5050 BNE L780; NOT ZERO PAGE
5060 L730 LDA TP
5070 CMP #2
5080 BNE L740
5090 LDA #16
5100 CLC
5110 ADC OP
5120 STA OP
5130 JMP TWOS
5140 L740 CMP #1
5150 BEQ L759
5160 CMP #3
5170 BEQ L759
5180 CMP #5
5190 BEQ L759
5200 L750 LDA #$32
5210 JSR P
```

```
5220 JMP L700
5230 L759 LDA #20
5240 CLC
5250 ADC OP
5260 STA OP
5270 ; ---------- SEE CHAPTER 11 FOR EXPLANATION OF THIS ERROR TRAP ------
5271 L760 LDA BUFFER+2,Y:CMP #89:BNE ML760; --- ERROR TRAP FOR LDA (15,Y)
5272 LDA OP:CMP #182:BEQ ML760; IS THE MNEMONIC LDX (IF SO, MODE IS CORRECT)
5273 JMP L680; IF NOT, JUMP TO MAKE IT (LDA $0015,Y) ABSOLUTE Y
5274 ML760 JMP TWOS
5275 ;-----------------------------------
5280 L780 LDA TP
5290 CMP #2
5300 BNE L790
5310 LDA #24
5320 CLC
5330 ADC OP
5340 STA OP
5350 JMP THREES
5360 L790 CMP #1
5370 BEQ L809
5380 CMP #3
5390 BEQ L809
5400 CMP #5
5410 BEQ L809
5420 L800 LDA #$33
5430 JSR P
5440 JMP L700
5450 L809 LDA #28
5460 CLC
5470 ADC OP
```

```
5480 STA OP
5490 JMP THREES;    END OF ADDR. MODE EVALUATIONS AND ADJUSTMENTS
5500 ;
5510 ;-------------------- ERROR REPORTING FOR DEBUGGING (PRINTS PC)
5520 P STA A; WHEN YOU INSERT A "JSR P" INTO YOUR SOURCE CODE, THIS ROUTINE
5530 STY Y; WILL PRINT THE PC FROM WHICH YOU JSR'ED.
5540 STX X;  AFTER AN RTS, THIS WILL REVEAL THE JSR ADDR.
5550 LDA #$BA; PRINT A GRAPHICS SYMBOL TO SIGNAL THAT THE PC IS TO FOLLOW
5560 JSR PRINT
5570 PLA; SAVE THE RTS ADDRESS (TO KEEP THE STACK INTACT)
5580 TAX
5590 PLA
5600 TAY
5610 TYA
5620 PHA
5630 TXA
5640 PHA
5650 TYA
5660 JSR OUTNUM; PRINT THE PC ADDRESS.
5670 LDA A; RESTORE THE REGISTERS.
5680 LDY Y
5690 LDX X
5700 RTS
5710 ;-------------------------
5720 CLEANLAB LDY #0; FILLS MAIN INPUT BUFFER ("LABEL") WITH ZERO. CLEANS IT.
5730 TYA
5740 CLEMORE STA LABEL,Y
5750 INY
5760 CPY #80
5770 BNE CLEMORE
5780 RTS
```

```
5790 ; ---------PRINT BRANCH OUT OF RANGE ERROR MESSAGE-------------
5800 DOBERR JSR PRNTCR; PRINT "BRANCH OUT OF RANGE" ERROR MESSAGE
5810 JSR ERRING
5820 JSR PRNTLINE; PRINT THE LINE NUMBER
5830 LDA #<MBOR; POINT "TEMP" TO THE ERROR MESSAGE "MBOR"
5840 STA TEMP; (MESSAGE BRANCH OUT OF RANGE, MBOR)
5850 LDA #>MBOR
5860 STA TEMP+1
5870 JSR PRNTMESS; PRINT THE MESSAGE
5880 JSR PRNTCR; PRINT A CARRIAGE RETURN AND
5890 JMP TWOS; BUNGLE AS AN ORDINARY 2-BYTE EVENT (TO KEEP PC CORRECT)
5900 ;-----------------------------
5910 .FILE EQUATE
```

Program 3-2. Eval, Apple Modifications

To create the Apple version of Eval, change the following lines in Program 3-1:

```
25 SETUP JMP EDITSU; START THE WEDGE
40 LDY #50
200 CMP #$A0
220 ;
230 ;
240 ;
310 CMP #$A0; IF NO SECOND BLANK SPACE
4282 SEC; SAVE THE LENGTH OF THE CODE
4283 LDA SA; FOR THE THIRD AND FOURTH
4284 SBC TA; BYTES OF THE BINARY
4285 STA LENPTR; FILE CREATED BY THE
4286 LDA SA+1; .D PSEUDOP
```

```
4287 SBC TA+1
4288 STA LENPTR+1
577Ø CPY #255
```

Program 3-3. Eval, Atari Modifications

To create the Atari version of Eval, change the following lines in Program 3-1:

```
1Ø :ATARI MODIFICATIONS--"EVAL"
3Ø TOP JMP EDIT
4Ø START LDA #Ø
5Ø STA 82
6Ø LDY #48
7Ø STRTLP STA OP,Y
8Ø DEY
9Ø BNE STRTLP
1ØØ LDA #<TOP
11Ø STA MEMTOP
12Ø STA ARRAYTOP
13Ø LDA #>TOP
14Ø STA MEMTOP+1
15Ø STA ARRAYTOP+1
16Ø LDA #1
17Ø STA HXFLAG
18Ø JSR CLRCHN
19Ø LDA RAMFLAG
2ØØ BNE SMORE
21Ø LDY #Ø
22Ø LDX ARGPOS
```

```
230 INX
240 STMØ LDA BABUF,X
250 CMP #155
260 BEQ STM1
270 STM3 STA FILEN,Y
280 INY
290 INX
300 JMP STMØ
310 STM1 STY FNAMELEN
320 JSR OPEN1
330 ;
340 ;
350 ;
470 LDA #160
4271 LDA SA
4272 STA LLSA
4273 LDA SA+1
4274 STA LLSA+1
4351 LDA RAMFLAG
4352 BNE OVEROPEN
4353 JSR OPEN1
4360 OVEROPEN JMP SMORE
4415 LDX #2:JSR CHKOUT:LDA #Ø:JSR PRINT:JSR
     CLRCHN
5550 LDA #160
5910 .FILE D:EQUATE.SRC
```

Chapter 4

Equate and Array:
Data Base Management

Equate and Array: Data Base Management

The job of setting up an array in machine language is simpler than you might imagine. The subprograms Equate and Array build and access a data base.

There are two basic ways to go about storing information: in *fixed* or in *variable length* fields. (A *field* in data base management means a single item, such as a single label name in LADS.) Fixed fields are easier to search, modify, and sort. Variable length fields save memory space. LADS uses variable length fields so the label table will take up as little space as possible.

A fixed field label system of managing data assigns a specified size in bytes for each item. If we had wanted to use this method of data storage for LADS' labels, we could have made a rule that label names cannot be larger than ten letters long. This would obviously make it simpler to manage the data.

However, then any label, even short labels, would always take up ten bytes. That would use up memory rather inefficiently. Instead, LADS allows labels to be of any length. If you are like me, the labels that you will think up naturally (without any restrictions imposed on your imagination) will normally average about five characters in length. Some will be longer, some shorter, but the average label will take up five bytes. Two bytes will be attached to each label to hold the integer number value which the label stands for. So, the average LADS variable (label name plus two-byte integer) takes up seven bytes. However, these variable length fields use up about 40 percent less memory when you consider that fields fixed at ten bytes would *always* take up ten bytes plus the two-byte number, never less.

Sons, Daughters, Clones

LADS itself is, of course, an ML program. You can have LADS object code assemble the LADS source code to disk or somewhere in RAM memory. This would create a new version of the assembler. If you'd made any changes to the source code, it would be an offspring, a son or daughter of LADS. If you didn't change the source code, you'd have created a clone, but the start address would differ.

LADS is about 5K long and uses 402 different labels. When it *assembles itself* from its own source code, it builds a label table which is 2851 bytes large. If it had fields fixed at ten bytes, the label table would be 4824 bytes large.

Why worry? It's true that the label table matters only during the actual assembly process. As soon as object code has been created and LADS returns to BASIC, the label table has served its purpose and can be tossed out like an eggshell after the egg is in the pan.

There are two good reasons for conserving memory: (1) the environment and (2) interactive freedom. Picture this: While assembling itself (or a comparably large program), LADS uses up about 8K of memory—5K for itself, perhaps 3K for the label table that builds down from the bottom of the assembler. And if you've chosen the option of assembling object code to RAM memory, add another 5K for the object code (the resulting ML program). A total of 13K. In some computers, this represents a significant bite out of the available memory.

What's more, LADS is supposed to be *interactive*. You are to have the psychological freedom you have with BASIC, to change things, to experiment, and then to quickly assemble and test the result. This means that you need space to write your source program (in RAM where a BASIC program is normally written). Perhaps you'll want a monitor extension in RAM too, like "Micromon" or "Supermon" or some other collection of ML utilities which permit single-step analysis of ML object programs, and other tools which are useful when debugging object code. And you might want "BASIC Aid" or "POWER" or some BASIC auto numbering, and other BASIC aids to manipulate the source code. You might want two different versions of your object code in RAM simultaneously so you can compare them in action.

The Programming Environment

All of these options require available RAM. If you can have them all in memory at once, you've got a better *environment* for developing an ML program. You won't always need to wonder if it's worth loading in a certain routine or utility: They're all there and ready to go. All your tools are at hand. This is a more efficient way to program. Tools that are out of reach are usually tools left unused.

Second, you want as few restrictions as possible when working with ML. You don't want to concern yourself about the length of each label name. Is it short enough? Does it duplicate a similar name? Eliminating these questions, too, is part of the interactivity, the mental freedom that comes with a smoothly running, efficient program development system. Variable length labels promote both effective memory conservation and an efficient programming environment.

Equate

The Equate subprogram starts off with one of those LDY #255 initializations. Remember that we don't always want to LDY #0 before a loop. There are times when the first event is the zeroth event. This is one of those times.

Line 40 sets Y to 255 so the INY in line 50 will make Y = 0. This allows us to LDA LABEL,Y and receive the first character in the buffer called LABEL. If we had set Y=0, the INY would have forced us to look at the *second* character in the buffer. Why not put the INY lower in the loop somewhere? That way, we would load in the first character the first time through the loop.

Obviously we can't INY just before the BNE in line 90. That would branch depending on the condition of Y itself, not on the item in A (which is our intention). For the same reason, we can't put it just before the BEQ in line 70. The only other safe place for it would be in a line between 70 and 80. That wouldn't do any damage to the branches because the CMP will reset the flags and the following BNE will act correctly.

This loop isn't moving characters from one buffer to another or anything. Its sole purpose is to count the number of characters in a label name, to find the length of the label. Y is the counter.

While locating Y in a line 75 would work correctly, it would be less clear what the loop is accomplishing. In cases like this, you have to decide where your personal priorities lie: Do you want to emphasize the function of a routine in a way that's more easily understood, or do you want to emphasize a uniform style of coding loops? If you prefer to always start such loops with LDY #0, by all means, go ahead. But that LDY #255 serves to alert you that this loop is a special kind of loop. If you come back later to modify a program, such signals can be helpful.

Once the length of our label is discovered, we add 2 to it by INY INY, to make room for the two-byte integer which will be attached to the label in our array. Each label stands for a number. And any legal number in ML can be stored within two bytes as an integer between 0 and 65535 ($0000–$FFFF).

Equate is called upon only during pass 1. On pass 1, the assembler puts each label into the array and attaches the two-byte integer onto the end of the word. So Equate's first job is to find out how much room to make in the array for each new label it comes upon. It makes room by lowering the MEMTOP variable by the length of the label name, plus two.

Building the Array Downward

SUBMEM moves our pointer down to make room for a new label. When SUBMEM is finished (**200**), the array is larger by the size of the new word we're adding to it, plus two bytes for the value of the word. The array is thus expanded, lowered.

Now we can store the label in the array. The first letter of each label in the array is special. It's *shifted*. That is, we add $80 (128 decimal) to the normal ASCII code value of the character. This is the same as setting the seventh bit.

If the label is "addnum," we want to store it as "Addnum" so that when we later search through the array, we can locate the start of each new label. The shifted letter will be our delimiter, separating the different labels. With fixed length fields, we wouldn't need a delimiter at all—each label would be exactly the same size as every other label. But our labels can vary in length, so we have to know where one begins and another ends.

The array will look like this (the xx is the two-byte value of each label):

AddnumxxSecondwordxxThirdwordxxFourthlabelxxFifhlabelxx

What exactly does it mean to say that a letter is *shifted*? In the ASCII code for alphabetic, numeric, punctuation (! or . or ,), and symbolic (# or % or *) characters, everything is assigned a code number which is lower than 128. Above 128 are the uppercase versions of letters, etc. Hence, above 128, the characters are *shifted*. For the purposes of ML, a shifted character is something with an ASCII code value greater than 127. It has the seventh bit set in its byte: 10000000. That leftmost bit would always be up in any shifted character. This phenomenon makes it easy to distinguish between shifted and unshifted characters. We can just LDA CHARACTER and then BMI (branch if seventh bit up) or BPL (branch if seventh bit down). The subprogram Array will make good use of this clue.

For now, all we want to do is shift the first character before we store it into the array. We just set up the seventh bit. If that's the same as adding $80 to a character, why not simply ADC $80 instead of EOR $80 (**230**)? With EOR we get a 1 if either of the compared bits is set. We get a 0 if both bits are 1 or if both bits are 0. The *only* way we get a 1 is if one of the bits is 0 and the other bit is 1. Any other situation results in a 0. Look at a bit comparison:

1 EOR 1 = 0
0 EOR 0 = 0
1 EOR 0 = 1

Consequently, EOR $80, with the $80 (binary 10000000) acting as a *mask,* will leave all the bits in the Accumulator unchanged, but *will* set the seventh bit. The main reason to use EOR is that we don't have to bother with clearing the carry (CLC) as we normally would prior to any addition.

After we store the shifted first letter in what is currently the lowest position in the array, we INY. This serves two purposes: It points us to the second character in the label word and also points us to the second space from the bottom of the array (where the second character of the label word belongs).

Address or Equate?

Now we load the second character and check if it's a space (**260–280**). We might be dealing with a one-character-long label, like P. We've got to check for this eventuality. Finding such a short label, we would jump down to see if there's an = sign. But if the label is more than one character long, we store the second letter in the array (**290**) and jump back up to fetch and store the third and any additional letters in the label name.

The essential thing to notice here is that a space is our delimiter in the buffer—letting us know when we've reached the end of the label word. And after finding a space, we are then prepared to distinguish between the two types of labels: PC and equate.

We compare the character following the space to $3D (this is the = sign). If it is an = sign, we branch to the routine which assesses the argument following the equals sign (is it hex? is it decimal?). Otherwise, we go through this BEQ to the routine which handles PC-type labels (*P*rogram *C*ounter types like: LABEL LDA 15, where the label indicates a location within the assembled program).

Storing the value of this kind of label is pretty simple: We just put the SA into the array. SA is the variable which always holds the current address during an assembly. But one thing remains to be done before we can return to the Eval subprogram to evaluate the LDA 15 part of this line. We've got to wipe out the word LABEL which precedes the LDA 15. Eval wouldn't know how to evaluate it. It's not a mnemonic.

After loading LABSIZE (the length of the label) into X, we load Y with 0. Y will point to the first space in the buffer, while X will count down until we've covered over the word LABEL (**430**).

Removing an Address Label

We load the leftmost part of the mnemonic/argument pair (the L of LDA is first), and we store it in the leftmost space in the buffer. In other words, the L of LDA covers up the L of LABEL. We continue with this process until we've loaded in a 0 and have therefore replaced LABEL LDA 15 with LDA 15, whereupon we store the final 0 as a delimiter and can return to Eval (**510**).

This next subroutine, NOAR (**520**), isn't in any sequential relationship to the other routines. It just happens to be here. It could be anywhere else in LADS just as easily. Its function is to ring the error bell and point TEMP to the message *NAKED LABEL* and then print that error message. It handles those cases when a programmer forgot to put anything after a label:

00 LABEL:INY

or

100 LABEL

or

100 LABEL =

Equate Labels

If we're not dealing with a PC-type label, though, we come here to store an *equate* label like LABEL = $22 (**590**) into the label array. We need to store Y first (in the variable LABPTR) so we can remember where in our array to put the value, the number following the equals sign. Remember that we've already stored the label name. What we need to do now is to put the value in the two bytes just following that name. When we arrive at this subroutine, Y is holding the correct offset from MEMTOP, the correct distance up in memory, from the bottom of the array to store the value.

There are now two possibilities. We are dealing with either a decimal number or a hex number. Hex numbers are translated by Indisk, the input subprogram, as they flow in from a disk file or RAM memory source code. So a hex number is already in the RESULT variable, waiting to be stored in the array.

But decimal numbers aren't translated as they come in. What's more, they arrive in ASCII form and must be converted into an integer by the subprogram Valdec.

We check the HEXFLAG to see if it's a hex number (**610**). If so, we can just put RESULT into the array and return to Eval (**750**).

But if it's a decimal number, we add the value of Y + 3 to the start-of-buffer address and point TEMP to the first character in the number we need to evaluate. We have to add this three to Y because the expression "space-equals sign-space" takes up three bytes. If we add this to the start of the buffer address, we're pointing to the first character in the number, pointing to the 1 in an example like: LABEL = 15.

Then we JSR to VALDEC, which looks at the number pointed to by TEMP and translates it from ASCII to an integer and puts the answer in the two-byte variable RESULT.

After this, we go through the same process as with hex numbers described above. The RESULT is transferred to the array, we pull off the two-byte RTS left on the stack (when we JSRed here from the Eval subprogram), and then jump back into Eval at INLINE, the place where a new line is pulled in from disk.

Array

The Array subprogram is essentially a search routine. It looks up a label's name in the array that was built by the Equate subprogram. When it finds a match, it puts the integer value of the array word into the variable RESULT. In effect, Array replaces a label with its number. Here's an example fragment of source code:

```
10 *= 864
100 NAME = 2
110 LABEL = 15
120 START LDA LABEL
```

On pass 1, Equate would store "Start864Label15Name02" into the array. The LADS label array builds down from the location of the start of LADS object code in memory. That is, the first part of LADS itself would be right above Name02. Line 120 contains two labels, *START* and *LABEL*. However, Equate ignores any labels which are not the first word in a given line. It only stores labels when it comes across the line in which they are *defined*. Any label being defined will be the first item in a given line. And if they are defined *twice* in the source code, that's an error.

(Note that, in the example of array storage above, Start864 is for illustration only. The number 864 is stored as a two-byte integer, not as 864, the ASCII characters we can read.)

While Equate ignores any label which is *not* the first thing on a line, Array ignores any label that *is* the first thing on a line. In the example above, Array would pay no attention to any of the labels except LABEL in line 120. It's Array's job to evaluate *expression labels*. An expression label is one that is used in an expression, one that is used as the *argument* of a mnemonic.

Array Works on Both Passes

Nevertheless, Array must operate on pass 1 as well as on pass 2. This is because pass 1 must keep an accurate PC, an accurate Program Counter. For Equate to store the correct number for labels, of the address (PC) type (like START in the example above), it must be able to find out precisely where in memory a given line is to be assembled. It must know that START is located at 864.

This problem derives from Zero Page addressing. LDA 15 takes up only two bytes in memory when assembled. LDA 1500 takes up three bytes. If labels were used in place of 15 and 1500 in these instructions, we must know whether to raise the PC by two or by three. So Array must look up all arguments on pass 1 to decide how much to increment the PC. (This PC, or Program Counter, is held in the LADS variable SA.)

In line 30 where Array begins, it moves the "bottom-of-LADS" (top of array) address from its permanent storage place, the variable ARRAYTOP, to the dynamic, changing pointer PARRAY. PARRAY will be lowered frequently as it points us down through the entire array.

Then we JSR to DECPAR which is the subroutine that lowers the PARRAY pointer by 1. And we stuff a $FF into the flag called FOUNDFLAG (**90**). This is a simple way to test if we've found our match. If we do find a match, as we'll soon see, we INC FOUNDFLAG. This means that FOUNDFLAG can more easily be tested in the way we want to test it. If it gets INCed once, it will be 0. INCed twice, it will be 1. INCed twice (or more) would mean that a label exists more than once in the array. That's an error, a *redefined label,* and we'll want to alert the programmer. Putting $FF into FOUNDFLAG thus allowed us to use BEQ to test for this error.

Checking for the Bottom

But all that comes later. The primary routine in Array starts with STARTLK (**100**), and oddly enough, the first thing we do is check to see if we're at the bottom of the array. The Equate subprogram always leaves the variable MEMTOP pointing to the bottom of the array. So, by subtracting our current position in the array (PARRAY) from the bottom of the array (MEMTOP), we can tell if we've finished looking through the array. If PARRAY is lower than MEMTOP, the carry will remain set, and we will then BCS down to the all-finished routine, ADONE.

Otherwise, we've got to keep on looking. Remember that Array must look through the entire array each time; even after it finds a match, it must continue looking for another match. This is the only way we can detect duplicated labels.

Array has to accomplish several things at once. It's got to point to the current position in the array, keep track of how large a given label is, and check each letter of each word. The chip registers will all be busy: A holds characters for checking, X keeps count of how large each label is, and Y (working with PARRAY) keeps track of our current position. Here, in line 160, we set X to zero.

Then we lower PARRAY by two to get past the number part of a label stored in array (**170–230**). We want to get past the 99 in /Label99/. Some of the stored numbers will have their seventh bit set; they'll be larger than 127. So we've got to jump over every stored number since the set seventh bit is our test to see if we've come upon the first character in a label name. We don't want numbers masquerading as label name delimiters.

At last we look at a character (**260**), and if the seventh bit is set, we BMI down to FOUNDONE. If it's not the start of a label name, we decrement PARRAY by 1 and jump up to LPAR to look at the next letter lower in memory within the array. Notice that we also raise the X (label length) counter (**320**). By the time we've found a shifted seventh bit indicating the start of a label name, X will hold the correct length of the name.

Double Decrement

Let's pause a minute to look at how a double decrement works (**280–310**). If, upon loading the low byte of PARRAY, the zero flag is set, we would be forced to lower the high byte of PARRAY (PARRAY+1 in line 300). If the low byte isn't yet lowered to zero, however, we can just lower the low byte and ignore the high byte (**310**). Note that a zero in the low byte requires lowering *both* the high and low bytes. Correctly decrementing $8500 would result in $84FF, lowering both bytes, while a correct decrement of $8501 would just lower the low byte: $8500.

Once we have located a set seventh bit, thus locating the start of a label name, we come to the FOUNDONE subroutine (**350**). Here we must first store PARRAY into the temporary holding variable PT so we can remember exactly where the label name begins. Then we reload A with the first character of the label (**390**) and compare it against the first character of the label we're looking for. That first character was previously in the variable WORK just before we came to Array from Eval.

If these first characters match, we go to LKMORE to check the rest of the word for a full match. If not, we go to STARTOVER.

In LKMORE, we first raise X to be the correct length of the current array label under examination. Then we save it in the variable WORK+1. We've got to save it at this point because now X will serve as the counter of the source label length. The *source label* is the word we're looking for, the label from the source code we're trying to find a match to.

The fact that some labels will be like (LABEL),Y or #LABEL (having a (or # as their first character) is a potential source of confusion to the Array search routine. To eliminate this confusion, whenever a (or # is encountered during the Eval subprogram, a special flag, BUFLAG, is raised. That makes it easy for us to skip over them here by raising the Y offset (**490**) if necessary.

Paradoxically, we simply INY again, right after this. That's because we want to point to the second character in the label (we got this far because the first characters matched). Nevertheless, the combination of INY and DECPAR (**490–500**) effectively takes care of the (or # situation and makes this INY point to the second letter of the label proper.

The LKM1 loop compares the entire rest of the source label against the array label (**520–600**). There are three ways, and only three ways, for us to get out of this loop. We can come upon a zero, which would surely be the end of the label in the buffer (the source label). A zero always means the end of a line of source code. Or we can come upon a character which is lower than 48. That includes things like left parentheses and commas in the ASCII code. Something like the comma in LDA LABEL,X would signal the end of the source label. (Checking for characters lower than 48, however, doesn't exclude numbers. We can still check for such legal labels as: LDA LABEL12.)

The Third Exit

The third way to exit this loop is when we fail to find a character match in the labels. Any point at which this happens, we "fall through" line 600—these characters do not BEQ, they're not equal. If they are equal, we go back up to check the next pair of characters. Notice that X continues to count the length of the words (**580**). In effect, it is counting the length of the *source* label (we already know the length of the array label and have it safely stashed away in the variable WORK+1).

If we leave this loop with a match, it will be a zero or a comma or right parenthesis *in the source label* that causes us to leave. X will then be holding the length of the source label. It's possible that we'll find an apparently "perfect match" which isn't, in fact, a match at all. For example, LABEL (as the array label) and LABE (as the source label) would appear to this

LKM1 loop as a perfect match. The only way we have of knowing that they do not really match is to compare their lengths.

If we fail to find a match, STARTOVER (**620**) just restores the correct array location of PARRAY (pointing at the first character in the label that just failed), and then we lower PARRAY by 1 (**660**) and jump back up to the STARTLK routine. STARTLK will also lower PARRAY by 1. This double lowering of PARRAY moves it past the number stored in the two bytes at the end of the next label down, thus preparing us to start the comparison process all over again.

On the other hand, if we *did* find a match, we go to FOUNDIT (**950**). Right off the bat, we check to see if the current value of X (length of the source label) matches the previously stored value of X (length of the array label). If they don't match, we've got that LABEL LABE situation, and we STARTOVER.

If everything checks out, though, we've got an authentic match. We raise the FOUNDFLAG. If this is the first match, FOUNDFLAG goes up from $FF to $00. That's fine. There should be *one* match. If, however, FOUNDFLAG is higher than 0, it means we've found more than one match, and we JSR to DUPLAB where the "duplicated label" error message is printed out (**1360**).

With or without this message, we next compensate for the (or # symbols which might be at the start of a source label and then load in the low byte of the number stored just above the array label. We put this byte into RESULT and put the high byte into RESULT+1. When we arrive here at FOUNDIT, the Y Register is pointing just past the end of the label. In other words, Y is pointing at the number stored with the label in the array. This is because we left the LKM1 loop when we got to the end of the label.

Pseudo-op Adjustments

Here's where we make the adjustments for two of our pseudo-ops: > < and +. If BYTFLAG is set, it means that < or > was used to request the low or high byte of a label. LDA #<LABEL requests the low byte (and Eval will only deal with low bytes in the # Immediate addressing mode). The label's low byte is already in the low byte of RESULT, so we need do nothing. But BYTFLAG is a special kind of flag. It has three states rather

than the normal two (set or clear, up or down) states. If it contains a 2, this signals that the #>LABEL pseudo-op was used, requesting the high byte of the label. To do this, we need to put the high byte of RESULT into the low byte of RESULT (**1140–50**). That's it.

PLUSFLAG signals a + pseudo-op like LDA LABEL+25. The amount we're supposed to add to LABEL (the 25) is already stored in the variable ADDNUM (by a subroutine in the Indisk subprogram). All we have to do here is add ADDNUM to the value in RESULT (**1180–1240**).

When these two pseudo-ops have been taken care of, we return to STARTOVER and keep looking for duplicated labels *if we're on pass 1.* On pass 1, we aren't allowed to leave the Array. On pass 2, however, it's not necessary to repeat this checking or to repeat the error messages, so we RTS, which sends us back to the Eval subprogram.

We've successfully put the value of the source label into RESULT. Now the Eval subprogram can go on to figure out the addressing mode, finish up by POKEing in the opcode and the argument, and then pull in the next line of source code.

But what if we didn't find any match to the source label and we've gone through the entire array? This can mean two things, depending on which pass we're on. On pass 1, it's harmless enough. It could well mean that the label hasn't yet been defined:

```
100 INY
110 BNE FORWARDLOOP
120 INX
130 FORWARDLOOP LDA 15
```

On the first pass, the label FORWARDLOOP will not be in the array until line 130. Nevertheless, the Array subprogram will search for it in line 110. And it won't find it. But so what? On pass 1, we can just ignore this failure to find a match and RTS back to Eval.

It would be a serious error, though, if the label could not be found in the array on pass 2. It would be an "undefined label" error.

When a Label Was Never Defined

Both of these possibilities are dealt with in the subroutine ADONE (**690–940**). If FOUNDFLAG has the seventh bit set, that means that it's still holding the $FF we put there at the very start of Array. We never found the match. We check the PASS, and if it's pass 2, we print the line number and the NOLAB error message "undefined label."

Then, no matter which pass it is, we still want to keep the program counter straight, or all the rest of the assembly will be off. The problem is that an undefined label doesn't give us the answer to the question: Is this a three-byte ordinary address or a two-byte zero page address? Is it LDA 15 or LDA 1500? Should we raise the PC by two or by three? If we raise it the wrong amount, any future reference to address-type labels will be skewed. Here's why:

```
100 *= 800
110 LDA LABEL; this label is undefined
120 ADDRESS INY; what is the location of ADDRESS here?
```

If LABEL is in zero page, ADDRESS = 802. If LABEL is not zero page, ADDRESS = 803. We should try to get this right on pass 1. Pass 2 depends on pass 1 for correct label values, including address-type labels. Even if a label is not yet defined, we should still try to raise the program counter by the correct amount.

In Eval there are routines called TWOS and THREES. TWOS raises the PC by two bytes for Zero Page and other two-byte-long addressing modes like LDA #15. THREES handles three-byte-long modes like Absolute addresses, etc. It's here in the Array subprogram, however, that we have to decide which of these routines to jump back to in Eval.

Branches like BNE and BEQ will often be undefined during pass 1 because the program is branching forward. We'll want to go to TWOS if there's an undefined label following a branch instruction. All branches are type 8, and we can easily check for them by LDA TP:CMP #8 (**860**). The other possible TWOS candidate is one of the > or < pseudo-ops. BYTFLAG signals one of them.

The # Immediate addressing mode is not tested for, so this adjustment isn't foolproof. The assumption is that any undefined label is essentially a fatal error and that there will have to be a reassembly. Most undefined labels are considered to be three-byte instructions and we JMP THREES (**920**).

This clarifies why LADS cannot permit the definition of a Zero Page address *within* the source code. All Zero Page address labels must be defined at the start of the source code, before any actual assembly takes place. Without this rule, our "yet-undefined-label" routine (**690–930**) will treat them, incorrectly, as three-byte address modes. It can recognize only branches and > < pseudo-ops as two-byte modes. Any other label that's not defined will be seen as a three-byte type.

Program 4-1. Equate

```
10 ; "EQUATE"  EVALUATE LABELS
20 ; COULD BE EITHER PC (ADDRESS) TYPE OR EQUATE TYPE.  STORE IN ARRAY.
25 ; FORMAT--NAME/2-BYTE INTEGER VALUE/NAME/2-BYTE VALUE/ETC...
30 ;-------------
40 EQUATE LDY #255; PREPARE Y TO ZERO AT START OF LOOP
50 EQ1 INY; Y GOES TO ZERO 1ST TIME THROUGH LOOP
60 LDA LABEL,Y; LOOK AT THE WORD, THE LABEL
70 BEQ NOAR; END OF LINE (SO THERE'S A NAKED LABEL, NOTHING FOLLOWS IT)
80 CMP #32; FOUND A SPACE, SO RAISE Y BY 2 AND SET LABEL SIZE (LABSIZE)
90 BNE EQ1; OTHERWISE, KEEP LOOKING FOR A SPACE.
100 INY
110 INY
120 STY LABSIZE
130 ;------------------- LOWER MEMTOP POINTER WITHIN ARRAY (BY LABEL SIZE)
140 SUBMEM SEC; SUBTRACT LABEL SIZE FROM ARRAY POINTER TO MAKE ROOM FOR LABEL
150 LDA MEMTOP
160 SBC LABSIZE
170 STA MEMTOP
180 LDA MEMTOP+1
190 SBC #0
200 STA MEMTOP+1;-------------
205 ;SHIFT 7TH BIT OF 1ST CHAR. TO SIGNIFY START OF LABEL'S NAME
210 LDY #0
220 LDA LABEL,Y
230 EOR #$80
240 STA (MEMTOP),Y; STORE SHIFTED 1ST LETTER
250 EQ3 INY
260 LDA LABEL,Y; IF SPACE, STOP STORING LABEL NAME IN ARRAY.
270 CMP #32
```

```
280 BEQ EQ2
290 STA (MEMTOP),Y; OTHERWISE, PUT NEXT LETTER INTO ARRAY &
300 JMP EQ3; CONTINUE.
310 EQ2 INY; NOW CHECK FOR = (SIGNIFYING EQUATE TYPE) (LABEL = 15)
320 LDA LABEL,Y
330 CMP #$3D; IF EQUATE TYPE, GO TO FIND ITS VALUE.
340 BEQ EQUAL
350 DEY; OTHERWISE, IT'S PC TYPE (LABEL LDA 15)
360 LDA SA; SO THE PC VARIABLE (SA) CONTAINS THE VALUE OF THIS LABEL
370 STA (MEMTOP),Y; STORE IT RIGHT AFTER LABEL NAME WITHIN ARRAY.
380 INY
390 LDA SA+1
400 STA (MEMTOP),Y
410 LDX LABSIZE; NOW, USING LABELSIZE AS INDEX, ERASE THE PC-TYPE LABEL
420 DEX;          FROM THE BUFFER.  FOR EXAMPLE, (LABEL LDA 15) NOW
430 LDY #0;       BECOMES (LDA 15).  THE LABEL NAME IS COVERED OVER
440 EQ5 LDA LABEL,X; TO PREPARE THE REST OF THE LINE TO BE ANALYZED
450 BEQ EQ4; NORMALLY BY EVAL.
460 STA LABEL,Y
470 INX
480 INY
490 JMP EQ5
500 EQ4 STA LABEL,Y
510 RTS; RETURN TO EVAL -------------------------
520 NOAR JSR PRNTCR:JSR PRNTLINE;NAKED LABEL FOUND (NO ARGUMENT) SO
525 JSR ERRING
530 LDA #<NOARG; RING BELL AND PRINT NAKED LABEL ERROR MESSAGE.
540 STA TEMP
550 LDA #>NOARG
560 STA TEMP+1
570 JSR PRNTMESS:JSR PRNTCR
```

```
580 JMP EQRET; RETURN TO EVAL----------------------
584 ;
585 ;--------------- HANDLE EQUATE TYPES HERE (LABEL = 15)
590 EQUAL DEY
600 STY LABPTR; TELLS US HOW FAR FROM MEMTOP WE SHOULD STORE ARGUMENT VALUE
610 LDA HEXFLAG; HEX NUMBERS ALREADY HANDLED BY INDISK ROUTINE, SO SKIP OVER.
620 BNE FINEQ; HEX FLAG UP, SO GO TO EQUATE EXIT ROUTINE BELOW.
630 INY; OTHERWISE, WE NEED TO FIGURE OUT THE ARGUMENT (LABEL = 15)
640 INY; THERE ARE THREE CHARS. ( = ) BETWEEN LABEL & ARGUMENT, SO
650 INY; INY THRICE.
660 STY WORK+1; POINT TO LOCATION OF ASCII NUMBER (IN LABEL BUFFER)
670 LDA #<LABEL; SET UP TEMP POINTER TO POINT TO ASCII NUMBER
680 CLC
690 ADC WORK+1
700 STA TEMP
710 LDA #>LABEL
720 ADC #0
730 STA TEMP+1
740 JSR VALDEC; CALCULATE ASCII NUMBER VALUE AND STORE IN RESULT
750 FINEQ LDY LABPTR; STORE INTEGER VALUE JUST AFTER LABEL NAME IN ARRAY
760 LDA RESULT
770 STA (MEMTOP),Y
780 LDA RESULT+1
790 INY
800 STA (MEMTOP),Y
810 EQRET PLA;PULL OFF THE RTS (FROM EVAL) AND JUMP DIRECTLY TO INLINE
820 PLA;       IGNORING ANY FURTHER EVALUATION OF THIS LINE SINCE EQUATE TYPE
830 JMP INLINE; LABELS ARE FOLLOWED BY NOTHING TO EVALUATE
840 .FILE ARRAY
```

For the Atari version of Equate, change line 840 to: **840 .FILE D:ARRAY.SRC**

Program 4-2. Array

```
10 ; "ARRAY" LOOKS THROUGH LABEL TABLE AND PUTS VALUE IN RESULT.
20 ;  (USED IN BOTH PASS 1 AND PASS 2)
30 ARRAY LDA ARRAYTOP;PUT TOP-OF-ARRAY VALUE INTO THE DYNAMIC POINTER (PARRAY)
40 STA PARRAY; IN OTHER WORDS, MAKE PARRAY POINT TO THE HIGHEST WORD IN THE
50 LDA ARRAYTOP+1; LABEL ARRAY
60 STA PARRAY+1
70 JSR DECPAR
80 LDA #$FF; SET UP FOR BMI TEST IF NO MATCH FOUND
90 STA FOUNDFLAG
100 STARTLK SEC; START LOOKING FOR LABEL NAME
110 LDA MEMTOP; CHECK TO SEE IF WE'RE AT THE BOTTOM OF THE ARRAY
120 SBC PARRAY
130 LDA MEMTOP+1
140 SBC PARRAY+1
150 BCS ADONE; IF SO, CHECK IF WE FOUND THE LABEL (OR FOUND IT TWICE)
160 LDX #0; SET LABEL NAME SIZE COUNTER TO ZERO
170 SEC; GO DOWN 2 BYTES IN MEMORY (PAST THE INTEGER VALUE OF A LABEL)
180 LDA PARRAY
190 SBC #2
200 STA PARRAY
210 LDA PARRAY+1
220 SBC #0
230 STA PARRAY+1
240 LDY #0
250 ;--------------------------
260 LPAR LDA (PARRAY),Y; LOOK FOR A 7TH BIT SET (START OF LABEL NAME)
270 BMI FOUNDONE; IF YES, WE'VE GOT TO THE START OF A NAME.
280 LDA PARRAY; OTHERWISE GO DOWN 1 BYTE IN ARRAY
290 BNE MDECX
```

```
300 DEC PARRAY+1
310 MDECX DEC PARRAY
320 INX; INCREASE LABEL NAME SIZE COUNTER
330 JMP LPAR
340 ;------------------------
350 FOUNDONE LDA PARRAY; WE'VE LOCATED A LABEL NAME IN THE ARRAY
360 STA PT; REMEMBER IT'S STARTING LOCATION
370 LDA PARRAY+1
380 STA PT+1
390 LDA (PARRAY),Y
400 CMP WORK; COMPARE THE 1ST LETTER WITH THE 1ST LETTER OF THE TARGET WORD
410 BEQ LKMORE; LOOK MORE CLOSELY AT THE WORD, IF 1ST LETTER MATCHED
420 JMP STARTOVER; IF IT DIDN'T MATCH, GO DOWN IN THE TABLE & FIND NEXT WORD.
430 ;---------------------------
440 LKMORE INX; RAISE LENGTH COUNTER BY 1
450 STX WORK+1; REMEMBER IT
460 LDX #1
470 LDA BUFLAG;THIS MEANS THAT # OR ( COME BEFORE THE NAME IN THE BUFFER
480 BEQ LKM1; IF THEY DON'T WE DON'T NEED TO RAISE Y IN ORDER TO IGNORE THEM
490 INY
500 JSR DECPAR; LOWER THE INDEX TO COMPENSATE FOR THE INY
510 ;
520 LKM1 INY
530 LDA BUFFER,Y; CHECK BUFFER-HELD LABEL
540 BEQ FOUNDIT; IF WE'RE AT THE END OF THE WORD (Ø), THEN WE'VE FOUND A MATCH
550 CMP #48; OR THERE'S A MATCH IF IT'S A CHARACTER LOWER THAN ASCII Ø (,OR+)
560 BCC FOUNDIT
570 ; NOT YET THE END OF THE "BUFFER" HELD LABEL
580 INX
590 CMP (PARRAY),Y; IF ARRAY WORD STILL AGREES WITH BUFFER WORD, THEN
600 BEQ LKM1; CONTINUE LOOKING AT THESE WORDS
```

```
610 ;---------------------------  NO MATCH, SO LOOK AT NEXT WORD DOWN -----
620 STARTOVER LDA PT; PUT PREVIOUS WORD'S START ADDR. INTO POINTER
630 STA PARRAY
640 LDA PT+1
650 STA PARRAY+1
660 JSR DECPAR; LOWER POINTER BY 1 (STARTLK WILL LOWER IT ALSO, BELOW VALUE)
670 JMP STARTLK; TRY ANOTHER WORD IN THE ARRAY
680 ;----------------------
690 ADONE LDA FOUNDFLAG
700 BMI AD1; DIDN'T FIND THE LABEL
710 RTS; ALL IS WELL.  RETURN TO EVAL.
720 AD1 LDA PASS
730 BNE AD1X; 2ND PASS-- GO AHEAD AND PRINT ERROR MESSAGE
740 BEQ ADONE1; ON 1ST PASS, MIGHT NOT YET BE DEFINED (RAISE INCSA/2S OR 3S)
750 AD1X JSR ERRING; LABEL NOT IN TABLE.  (TREAT IT AS A 2-BYTE ADDRESS)
760 JSR PRNTLINE
770 JSR PRNTSPACE
780 LDA #<NOLAB
790 STA TEMP
800 LDA #>NOLAB
810 STA TEMP+1
820 JSR PRNTMESS; RING BELL AND PRINT NOT FOUND MESSAGE
830 JSR PRNTCR
840 ADONE1 PLA
850 PLA;
860 LDA OP
870 AND #31
880 CMP #16
890 BEQ ADO2; CHECK IF BRANCH INSTRUCT.
900 LDA BYTFLAG
910 BNE ADO2; < OR > PSEUDO
```

```
920 JMP THREES
930 ADO2 JMP TWOS
940 ;
950 FOUNDIT CPX WORK+1;CHECK LABEL LENGTH AGAINST TARGET WORD LENGTH
960 BEQ FOUNDF; THEY MUST EQUAL TO SIGNIFY A MATCH. (PRINT/PRIN WOULD FAIL)
970 JMP STARTOVER; FAILED MATCH
980 FOUNDF INC FOUNDFLAG; RAISE FLAG TO ZERO (FIRST MATCH)
990 BEQ FOFX; IF HIGHER THAN 0, PRINT DUPLICATION LABEL ERROR MESSAGE
1000 JSR DUPLAB
1010 FOFX LDY WORK+1
1020 LDA BUFLAG; COMPENSATE FOR # AND (
1030 BEQ FOF
1040 INY
1050 FOF LDA (PARRAY),Y; PUT TABLE LABEL'S VALUE IN RESULT
1060 STA RESULT
1070 INY
1080 LDA (PARRAY),Y
1090 STA RESULT+1
1100 LDA BYTFLAG
1110 BEQ CMPMO; IS IT > OR < PSEUDOPRINT
1120 CMP #2
1130 BNE AREND
1140 LDA RESULT+1; STORE HIGH BYTE INTO LOW BYTE
1150 STA RESULT
1160 CMPMO LDA PLUSFLAG; DO ADDITION + PSEUDO OP
1170 BEQ AREND
1180 CLC; ADD THE + NUMBER "ADDNUM" TO RESULT
1190 LDA ADDNUM
1200 ADC RESULT
1210 STA RESULT
1220 LDA ADDNUM+1
```

```
1230 ADC RESULT+1
1240 STA RESULT+1
1250 AREND LDA PASS; ON 2ND PASS, CHECK FOR DUPS
1260 BNE ARENX
1270 RTS; GO BACK TO EVAL
1280 ARENX JMP STARTOVER; ON PASS 2, LOOK FOR DUPS (SO CONTINUE IN ARRAY)
1290 ;------------------------------
1300 DECPAR LDA PARRAY; LOWER ARRAY POINTER BY 1
1310 BNE MDEC
1320 DEC PARRAY+1
1330 MDEC DEC PARRAY
1340 RTS
1350 ; ----------------
1360 DUPLAB JSR ERRING; RING BELL AND PRINT DUP LABEL MESSAGE
1370 LDA #<MDUPLAB
1380 STA TEMP
1390 LDA #>MDUPLAB
1400 STA TEMP+1
1410 JSR PRNTMESS
1420 JSR PRNTCR
1430 RTS;-----------------------------------
1440 .FILE OPEN1
```

For the Atari version of Array, change line 1440 to:

1440 .FILE D:OPEN1.SRC

Chapter 5

Open1, Findmn, Getsa, and Valdec:

I/O Management and Number Conversions

Open1, Findmn, Getsa, and Valdec:
I/O Management and Number Conversions

I/O (Input/Output), a computer's method of communicating with its peripherals, is one of the most machine-specific and potentially complex aspects of machine language programming.

Sending or receiving bytes to or from disk or tape drives and sending bytes to a printer are the most common I/O activities. A large part of a computer's ROM memory is usually devoted to managing I/O.

I/O is machine-specific because each manufacturer invents his own way of managing data, his own variations on the ASCII code, and his own disk or tape operating systems.

And I/O is complex because printers and disk and tape drives differ greatly in such things as how fast they can store bytes, how many bytes they can accept, and esoteric matters like timing, error checking, and special control signals.

ML programmers are frequently advised to perform I/O operations in BASIC and then SYS, CALL, or USR into the ML after the hard part has been accomplished by the computer's operating system. This works well enough with small ML projects. But it can become awkward in a large ML program. LADS itself must open and close disk files pretty often. It would be inefficient to require LADS to fly down into an attached BASIC program for this. Also, large ML programs are easiest to save, load, and use if they are written *entirely* in ML.

Fortunately, we can access BASIC's ROM routines from within an ML program. Certain registers and pointers in zero page need to be set up, then we can JSR to open a file to a peripheral. After that, we can send or receive bytes from that file.

Since these routines *are* so machine-specific, we'll look at the Commodore techniques in this chapter. See Appendix C for an explanation of the Atari and Apple I/O techniques.

Commodore I/O

Some peripherals are intelligent and some are dumb. Commodore disk drives are highly intelligent—they've got large amounts of RAM and ROM memory. One consequence of this is that relatively little I/O computing needs to be done within the computer proper. A Commodore disk drive is a little computer itself. You can just send it a command, and it takes over from there.

The tape drives, though, are dumb. ROM intelligence within the computer must manage I/O to tape. Some printers aren't so dumb, but since you can choose from so many different models and brands, the computer just sends out a sequence of raw bytes when you print to a printer. Your BASIC or operating system makes no effort to control fonts, formatting, or any other special printer functions. You are expected to send any necessary printer control codes via your software. If the printer is equipped to TAB or justify text, that's up to the printer's ROM.

Open1

In the subprogram Open1, there are four Commodore-specific subroutines. In many respects, they are identical subroutines. Each opens a file to an external device in much the same way. Only the specifics differ. The first subroutine, OPEN1, starts communication with a disk file which will be *read*. That is, the source code will come streaming in from this file so that LADS can assemble it. This file will be referred to as file 1.

The second subroutine, OPEN2, opens file 2 as a *write* file. If the user includes the .D NAME pseudo-op within his source code, the results of a LADS assembly, the object code, will be stored on disk in a file called NAME. OPEN2 makes the disk create this file.

The third subroutine, OPEN4, creates a simple write file to the printer. It, too, is similar to the others except that there is, of course, no filename.

Looking at OPEN1, the first event is a call to the CLRCHN subroutine within BASIC. All I/O (including that to the screen and from the keyboard) is governed by this opened-files concept in Commodore computers. The normal I/O condition is output to the screen and input from the keyboard. CLRCHN sets the computer to this condition. It is a necessary preliminary before any other opening or closing of files.

Resetting the Disk Program Counter

Next we close file #1 (**50–60**). This resets the disk intelligence. As we shift from pass 1 to pass 2, we've been reading through file #1 to bring in our source code. On pass 2, we want to start all over again with the first byte in the disk source file. It is necessary to close, then reopen, file #1 to force the disk intelligence to again point to that first byte in the file.

Next we must prepare some zero page file-manipulation pointers. We store the file number to FNUM, the device number (8 is the disk device number in Commodore computers) to FDEV, and the secondary address to FSECOND. All of this is precisely what we do in opening a file from BASIC with OPEN 1,8,3.

Then we have to point to the location of the filename within RAM. LADS holds filenames in a buffer called FILEN, so we put the low and high bytes of FILEN's address into the FNAMEPTR. Then, at last, we go to OPEN, the BASIC subroutine which opens a disk file.

The four zero page locations and the OPEN routine in ROM are all machine-specific. They are defined in the Defs subprogram. OPEN2 is identical except for a different filename, a different file number, and a different secondary address (which makes it a write file).

OPEN4, too, is identical except that the secondary address is ignored, the device number is 4 (for printers in Commodore computers), and there is no filename.

Line 430 reveals a fifth zero page location which must be POKEd before calling the OPEN subroutine in BASIC ROM. It holds the length of a filename. (Opening to a printer uses no filename, so a zero is put into FNAMELEN [**430**].)

Both of the other subroutines, OPEN1 and OPEN2, do not need to POKE FNAMELEN. It is POKEd just before LADS JSRs to either of them.

LOAD1, the final I/O subroutine in this subprogram, is used with the assemble-from-RAM-memory version of LADS. In this case, the source code files are LOADed into RAM before they are assembled. This means that we need to imitate a typical BASIC LOAD of program files.

The LOAD subroutine within BASIC requires that the LOAD/VERIFY flag be set to LOAD (rather than VERIFY), that 8 be declared the device (disk), and that the name of the program to be loaded be pointed to. Then the machine-specific

LOAD routine within BASIC is called. After that, the program (the source code) is loaded into the normal RAM address for BASIC programs.

Findmn: Table Lookup

This subprogram is similar to the Array subprogram: Both look through an array and find a match to a "source" word. Yet Findmn is simpler than Array. It doesn't need to check for word lengths. Also, the numbers (the *values*) associated with the words in the array are more simply retrieved. Findmn tries to find a mnemonic like LDA or BCC in a table of all 56 of the 6502 machine language mnemonics.

This table (or array) of mnemonic names is in the subprogram Tables at the very end of LADS source code. The mnemonics table starts off like this:

```
50 MNEMONICS .BYTE "LDALDYJSRRTSBCSBEQBCCCMP
60 .BYTE "BNELDXJMPSTASTYSTXINYDEY
```

and continues, listing all of the mnemonics.

This array of mnemonics is simpler and faster to access than our array of labels because it's what's called a *lookup table*. It has four characteristics which make it both easy to access and very efficient: It's a fixed field array (all items are three bytes long), it's static, it's parallel, and it's turbo-charged.

Charles Brannon, my colleague at COMPUTE! Publications, is a proponent of what he calls "turbo-charged code." He writes an ML program, gets the logic right, and then takes a cold look at things, especially at heavily used loops. Is the first CMP the one most often true in a series of CMPs? Or would it be faster to rearrange these CMPs in order of their probability of use? Should an Indirect Y addressing mode be replaced by an even faster structure such as self-modifying Absolute addressing? Would a lookup table be a possible replacement for some computed value? Sometimes, small changes can result in extraordinary gains in speed. For example, after LADS was finished and thoroughly tested, it took 5 minutes, 40 seconds to assemble itself (5K of object code).

A cold look, about five hours of work, and the resulting few minor changes in the source code brought that time down to its present speed for self-assembly: 3 minutes, 21 seconds. (This speed test was conducted with only the .D name pseudo-op activated, on a Commodore PET/CBM 8032, with a 4040 disk drive, and involving far fewer comments than found with the source code as published in this book. The use of additional pseudo-ops, additional comments, or other computer/disk brands and models will result in different assembly speeds. The Apple has a faster disk drive, for example, and the LADS Apple version is even faster than the Commodore version.)

How does this mnemonics lookup table differ from the label array? They're both arrays, but the label array is a *dynamic* array. It changes each time you reassemble different source code. A lookup table, by contrast, is static: It never changes. It's a place where information is permanent and lends itself, therefore, to a bit of fiddling, a bit of turbo-charging.

A Special Order

First of all, in what order did we put these mnemonics? They're not in alphabetical order. In that case, ADC would be first. They're not in the numeric order of their opcodes either. Using that scheme, BRK would be first, having an opcode of 0. Instead, they're in order of their frequency of use in ML programming. The order wasn't derived from a scientific study—I just looked at them and decided that I used LDA more often than anything else. So I put it first.

The reason for putting them in order of popularity is that every line of source code contains a mnemonic. Every time a mnemonic is detected, it must be looked up. Since this lookup starts with the first three-letter word in the table (all mnemonics are three letters long) and works its way up the table, it makes sense to have the most common ones lowest in the table. They'll be found sooner, and LADS can continue with other things. It turns out that rearranging the order of the mnemonics in the table resulted in an increase in speed of considerably less than 1 percent, but everything helps. The principle is valid, even if it doesn't accomplish much in this case.

The second quality of a lookup table—parallelism—*is* rather significant to the speed of LADS. Right below the MNEMONICS table in the Tables subprogram are two parallel tables: TYPES and OPS. (See the Tables subprogram at the end of Chapter 9.) TYPES can be numbers from 0 to 9. It is handy to group mnemonics into these ten categories according to the addressing modes they are capable of using. Some mnemonics, like RTS, INY, and DEY, have only one possible addressing mode (they take no argument and have *Implied* addressing). They are all labeled type 0. The branching instructions, BNE, BEQ, etc., are obviously related in their behavior as well: They are type 8. This categorization helps the Eval subprogram calculate addressing modes. This table of TYPES *parallels* the table of MNEMONICS. That is, the first mnemonic (LDA) is type 1, so the number 1 is the first number in the table of TYPES. The fifth mnemonic in the MNEMONICS tables, BCS, is paralleled by the fifth number in the TYPES table, 8.

The Efficiency of Parallel Tables

What's the value of putting them in parallel? It allows us to use the Y or X Register as an index to quickly pull out the values in any table which is parallel to the primary lookup table, MNEMONICS. Once we've found a match within MNEMONICS, we can simply LDA TYPES,X to get that mnemonic's type. And we can also LDA OPS,X to get the opcode for that mnemonic. All this works because we INX after each failure to match as we work our way up through the MNEMONICS table. X will point to the right item in each of the parallel tables, after we find a match.

But now on to the actual lookup techniques which are used in the Findmn subprogram. As usual, we set our index counters, X and Y, before entering a loop. X gets $FF (**40**), so it will zero at the first INX at the start of the loop. Y gets 0. You can tell that this was the first subprogram written in LADS. Nowhere else can we achieve the elegant simplicity of calling a loop LOOP and the end of the routine END (**390**). After using them once, we'll have to come up with other names for loops and exits.

Anyway, we enter LOOP and look at the first character in the MNEMONICS table (**60**). If it matches the first character in the buffer LABEL (holding something like: LDA 15), we jump down to look for a match to the second, and then the final, character in the mnemonic. Otherwise, if there is no match, we INY INY INY to move up three characters in the MNEMONICS table and prepare to compare the first letter of the second mnemonic against our source mnemonic.

When looking something up, it saves time if you just test first characters before going on to whole-word tests.

Assuming a first characters match, MORE (**150**) compares the second characters. If they match, we go on to MORE1. This time a failure to match results in two INYs because there was one INY at the start of MORE. MORE1 tests the third characters. If it fails, we only need one INY. In each case, a failure returns to LOOP. LOOP itself fails when it has exhausted all 56 mnemonics in the table and no match has been found. Since each attempt causes X in INX, we can test for the end of the table of 56 mnemonics by CPX #57 (**120**).

If we have exhausted the table, we jump back into the Eval subprogram where label definitions are evaluated. Since we didn't find a mnemonic as the first thing on a source code line, it must be a label like:

100 LABEL LDA 15

or

100 LABEL = 75

JMP for JMP

Note that we don't need to PLA PLA the return address of an RTS off the stack before JMPing back to Eval from this subprogram. That's because we JMPed here from Eval. Both possible returns to Eval will be JMPs. That makes it possible for us to JMP directly to Findmn from Eval. For speed, we can JMP back to two different places within Eval, depending on whether we did or did not find a mnemonics match.

Finding a match, however, sends us to the FOUND subroutine (**300**) where we check to see if there is a blank character or a zero (end of line) following the supposed mnemonic. If there isn't, that means we've got a label which *looks* like a mnemonic: INYROUTINE or BPLOT or something. We can't let that fool us. If there's a character in the fourth position, such words reveal themselves to be labels. If so, we go back to Eval via NOMATCH.

But let's say that all was well. It's not an address label, it's not an equate label, it's not a label disguised as a mnemonic. We've located a true mnemonic. All we have to do is pick its TYPE and OPCODE out of their tables and store them in their holding places, the variables TP and OP, and JMP back to EVAR in Eval. EVAR is a subroutine in Eval which examines the argument of a mnemonic to determine its addressing mode.

Getsa: The Simplest Routine

This subprogram has only one mission: to point to the starting address in the source code program. Here's what it points to:

10 *= 864

Getsa pulls off the first six bytes (in a Commodore disk program file) so that it can check to see if the seventh byte is the * character (**120**). If so, Getsa returns to the calling routine in Eval (**200**). If not, it prints the NO START ADDRESS error message and goes to FIN (**190**), the shutdown (return to BASIC) routine.

Conditional Assembly

There are two fundamentally different versions of LADS. The version presented as object code (to be typed in) in this book assembles from *disk-based* source code. You create BASIC-like "programs" on disk, and then LADS reads them and assembles them without bringing any source code into RAM memory.

An easy modification to LADS, however, will allow it to assemble directly from source code within RAM memory. A few trivial changes to LADS' own source code and you can assemble a new, memory-based LADS. These changes are described between lines 430 and 640 of the Getsa source code printed at the end of this chapter. The changes are described in greater detail in Chapter 11, "Modifying LADS."

But this Getsa source code illustrates one way that your source code program can *conditionally assemble.* Notice line 210. The MEMSA and CHARIN routines below it will never be assembled. When LADS sees the .FILE pseudo-op, it will immediately turn its attention to the Valdec source code. .FILE shuts down the current file and switches to the named source file, *ignoring any additional source code in the current file.*

Thus, to assemble the "conditional" part of this source code, all you have to do is move .FILE *below* the new source code. See the instruction in line 580 of this Getsa subprogram. That's how you do it to create a memory-based version of LADS.

Another way to conditionally assemble is to insert the .NO pseudo-op, thus turning off object-code-to-memory-storage until the .O pseudo-op turns it back on. You could write your own .ND (no storage to disk) pseudo-op if you want to control assembly which is sending its object program to a disk drive. Another pseudo-op you could write would be something like .NA for No Assembly which would cause LADS to simply search down through source code (taking no actions other than building the label array) until it located a .A pseudo-op, turning all assembly back on. These .ND, .NA, and .A pseudo-ops aren't built into LADS, but would be easy to add if you felt you'd have a use for them.

Valdec: Number Conversion

Numbers such as the 15 in LDA 15 are held in ASCII code format within source programs. In other words, when LADS pulls in the 15, it doesn't get the *number* 15. It gets 1-5 instead. It gets the ASCII for 1 and the ASCII for 5: 49 and 53 decimal. (As an aside, 1 and 5 are $31 and $35 in hex. It's pretty easy to mentally convert ASCII hex to numeric form. Just drop the leading 3 from any hex ASCII number.)

What Valdec must do is turn 49 53 into the two-byte number 0F 00 which the computer can recognize and work with. This is just a bit more complicated than it might seem. The complexity comes from the fact that the 1 in 15 is really 10 times 1. The Valdec subprogram which handles this ASCII-to-integer translation will have to multiply by 10,000 or 1000 or 10 or 1—depending on the position of the ASCII digit. We don't need to worry about numbers higher than 65535 since ML doesn't often need to calculate higher than that. All addresses that the 6502 chip can reach are within that range, and two bytes cannot hold a larger number anyway. Therefore, multiplication by 10,000 will take care of any case we might come across.

And since 10,000 is just 10 × 10 × 10 × 10, we'll really only need a way of multiplying by 10 a maximum of four times. So all that's really needed is a multiply-by-10 routine that we can loop through as often as necessary. Lines 400–550 perform this operation.

But let's start at the start. Anything in LADS which calls upon Valdec for its services will have already set up the TEMP pointer to point to the first ASCII character in the number to be translated. Also, the number will end with a 0 delimiter. (This isn't the ASCII 0, which is $30. It's a true zero.)

Determining Length

After Valdec finishes, it leaves the results in the two-byte register called RESULT.

First Valdec finds the length of the ASCII number (**50–90**). Our example number, 15, would be two bytes long. Its length is stored in the variable VREND, and we then clean out the RESULT register by storing 0 into it (**130–150**). Then X (not the register, the variable) is stuffed with a 1 (**170**) so it can tell us how many times to loop through the times-ten routine for each digit. As we move from right to left, reading first the 5 then the 1 in 15, X will be raised. Coming upon the 5, X will be 1, and we'll perform no multiplication. The first thing the loop for multiplication does is DEX, so 1 becomes 0 and we exit the loop (**250**).

Coming upon the 1, X will tell us to go through the timesten routine once. In other words, we multiply 1 times 10 for a result of 10. This, added to 5, gives the 15 we're after.

But let's back up to where we were, at VALLOOP (**180**). We can take advantage of the fact that the ASCII code was designed so that the lower four bits in each ASCII numeral byte hold the actual number: $35 stands for 5. How do we extract the number $05 from $35? We could subtract $30. Even simpler is AND #$0F. AND turns bits off. Wherever a bit is off in the mask (the #$0F in this example), the bit will be off in the result:

```
     $35          (ASCII for 5)
AND  0F           (the four high bits are all off,
                  the four low bits are on—they
                  have no effect)
     $05          (the answer we're after)

     00110101     ($35, prepared to be stripped of its high bits by)
AND  00001111     ($0F, the mask, turning bits off where the 0's
     ________     are)
     00000101     ($05, leaving the number we want)
```

Here we load in the rightmost character, the 5 in 15, the $35 in $31 $35. And strip off the 3, leaving the 5. Then that's stored in two temporary variables: RADD and TSTORE. Next we fill both of the high bytes of these variables with 0 (**220–240**). That makes them officially correct. Nothing lingers in their high bytes to confuse things later when we perform two-byte addition.

Now that our digit 5 is safely tucked away, we need to multiply it by 10 as many time as necessary. DEX lowers X. With this first character, X becomes 0, and we BEQ to the exit (**330**). When we come through this loop next time, holding the 1 in 15, X will become 1 and we'll therefore JSR TEN (**270**) one time, making 1 into 10.

Keeping Track of Position

After the subroutine TEN has multiplied the number in RADD (named for Result of ADDition) by 10, we transfer the result from RADD over to TSTORE (**280–310**). Why the transfer? Because in the 100's position, a digit would need to be multiplied by 10, twice. The 2 in 215 would have to be 2 times 10 times 10. So TSTORE has to keep a running total of the results achieved by the TEN subroutine. TEN uses RADD during multiplication. Obviously, a second two-byte variable will have to keep track of the total as, more than once, we multiply the larger digits by 10.

Another running total, the result of all Valdec's efforts, is kept in the variable RESULT. That will ultimately hold our final answer. But each time we achieve an interim answer on a single digit, we JSR VALADD (**350**) to add the results of that digit's multiplication to RESULT (**570–640**).

Meanwhile, back up at line 360, we DEY to point to the next higher digit, the digit next to the left. And DEC VREND to see if we've reached the end of our ASCII number and cannot RTS. If not, we go back up and load in the next digit, continuing to add to the running total in RESULT.

The multiply-by-ten routine called TEN (**410**) is worth a brief examination. Let's imagine that we have put a 1 into RADD (**200**) and we're going through the TEN loop once, multiplying it by 10. We clear the carry. ASL shifts each bit in RADD (the low byte of this two-byte number) to the left by 1. The interesting thing is that the seventh bit goes into the carry. Then we ROL RADD+1, the high byte, which *rotates* each bit to the left. This is the same as the ASL shift to the left. The seventh bit pops into the carry. But with ROL, the *carry moves into the zeroth bit.* A combination of ASL ROL shifts all the bits in a two-byte number to the left by 1:

Carry bit	*high byte*	*low byte*	
0	00000000	00000001	(our 1 before ASL low byte, ROL high byte)
0	00000000	00000010	(after)

You can see that this, in effect, multiplies these bytes by 2. If we ASL/ROL again, we get:

0	00000000	00000100	(the original number, multiplied by 4)

At this point, our answer is 4. We've multiplied the original 1 by 4 with an ASL/ROL combination, performed twice.

Now we CLC again and add the original number (1) to the current result (4), giving us 5 (**460–520**). It's easy to see that all we need to do now is one more ASL/ROL, which multiplies the running total by 2 one more time:

	Carry bit	*high byte*	*low byte*	
	0	00000000	00000100	(4)
+	0	00000000	00000001	(added to the original 1, gives)
	0	00000000	00000101	(5)

then, we just ASL the low byte:

0	00000000	00001010	(10)

ROL the high byte (which has no effect on this small a number):

0	00000000	00001010	(giving us 10)

That final ASL/ROL multiplies 5 times 2, and we've got the right answer (**530–540**). This trick—multiply by 4, add the original number, multiply by 2—will work whenever you need to multiply a number by 10. Other combinations will multiply by other numbers. And as Valdec illustrates, you can calculate powers of 10 by just running the result through this TEN subroutine as often as necessary.

Program 5-1. Open1, Commodore

```
10 ; "OPEN1"  OPEN 1,8,3,"WHATEVER NAME FROM SCREEN"
20 ; OPEN A FILE ON DISK (THIS TYPE OF FILE IS READ FROM)
30 ;------------------
40 OPEN1 JSR CLRCHN;RESTORE NORMAL I/O (OUTPUT TO SCREEN, INPUT FROM KEYBOARD)
50 LDA #1; CLOSE DOWN DISK FILE CHANNEL #1
60 JSR CLOSE; (WE'RE GOING TO REOPEN IT NOW, BUT WE CLOSE IT FIRST)
70 LDA #1
80 STA FNUM;              FILE#
90 LDA #8
100 STA FDEV;              DEVICE NUMBER.
110 LDA #3
120 STA FSECOND;                SECONDARY ADDR.
130 NAMEAD LDA #<FILEN; SET POINTER TO FILE NAME BUFFER (FILEN) IN LADS.
140 STA FNAMEPTR ;POINTER TO FILENAME ADDR.
150 LDA #>FILEN
160 STA FNAMEPTR+1
170 JSR OPEN; ROUTINE WITHIN BASIC THAT OPENS UP A NEW FILE
180 RTS
190 ;------------------------
200 ; OPEN 2,8,2,"NAME"  (OPENS DISK PROGRAM FILE FOR WRITING OBJECT CODE)
210 ;------------------------
220 OPEN2 LDA #2; SEE DEFINITIONS ABOVE (SAME SETUP)
230 STA FNUM
240 LDA #8
250 STA FDEV
260 LDA #2
270 STA FSECOND
280 LDA #<FILEN
290 STA FNAMEPTR; POINTER TO FILENAME ADDR.
```

```
300 LDA #>FILEN
310 STA FNAMEPTR+1
320 JSR OPEN
330 JSR CLRCHN
340 RTS
350 ;-------------------------
360 ; OPEN 4,4              (OPENS FILE TO PRINTER)
370 ;------------------------
380 OPEN4 LDA #4; SAME FORMAT, EXCEPT FNAMELEN
390 STA FNUM
400 LDA #4
410 STA FDEV
420 LDA #0; THERE IS NO FILE NAME SO SET FILENAME LENGTH TO ZERO.
430 STA FNAMELEN
440 JSR OPEN
450 JSR CLRCHN
460 RTS
470 ;-------------------------
480 ; LOAD "NAME"  (LOADS A PROGRAM FILE, A SOURCE CODE FILE INTO RAM)
490 ;------------------------
500 LOAD1 JSR CLRCHN;RESTORE NORMAL I/O
510 LDA #0
520 STA LOADFLAG; LOAD/VERIFY FLAG
530 STA ST; THE STATUS BYTE
540 LDA #8
550 STA FDEV;             DEVICE NUMBER.
560 LDA #<FILEN; SET POINTER TO FILENAME BUFFER (FILEN) IN LADS.
570 STA FNAMEPTR ;POINTER TO FILENAME ADDR.
580 LDA #>FILEN
590 STA FNAMEPTR+1
600 JSR LOAD; ROUTINE WITHIN BASIC THAT LOADS IN A PROGRAM
```

```
    JSR CLRCHN
615 LDA RAMSTART:STA PMEM:LDA RAMSTART+1:STA PMEM+1
620 RTS
630 .FILE FINDMN
```

Program 5-2. Open1, Apple

```
5 ;OPEN INPUT FILE
10 OPEN1 JSR CLRCHN
20 LDA #1; CLOSE FILE IF ALREADY OPEN
30 JSR CLOSE
40 LDA #<OPNREAD
50 STA FMOP
60 LDA #>OPNREAD
70 STA FMOP+1
80 JSR FMDRVRO
90 INC FOPEN1; SET INPUT FILE TO OPEN
100 RTS
105 ; OPEN OUTPUT FILE
110 OPEN2 LDA #<OPNWRIT
120 STA FMOP
130 LDA #>OPNWRIT
140 STA FMOP+1
150 JSR FMDRVRO
160 INC FOPEN2; SET OUTPUT FILE OPEN
170 RTS
180 OPEN4 RTS; OPEN NOT NEEDED TO PRINTER
185 ; READ ONE BYTE FROM INPUT FILE
190 RDBYTE LDA #<RD1B
200 STA FMOP
210 LDA #>RD1B
```

```
220 STA FMOP+1
230 JSR FMDRVR
240 JSR $3DC
250 STA PARM+1
260 STY PARM
270 LDY #08
280 LDA (PARM),Y; GET THE BYTE
290 RTS
295 ; WRITE ONE BYTE TO OUTPUT FILE
300 WRBYTE STA WRDATA
310 LDA #<WR1B
320 STA FMOP
330 LDA #>WR1B
340 STA FMOP+1
350 JSR FMDRVR
360 RTS
365 ; CLOSE INPUT FILE
370 CLOSE1 LDA FOPEN1; CHECK TO SEE IF INPUT FILE IS OPEN
380 BEQ CLOSE4; IF NOT EXIT
390 LDA #<CLOSER
400 STA FMOP
410 LDA #>CLOSER
420 STA FMOP+1
430 JSR FMDRVR
440 LDA #0
450 STA FOPEN1; SET INPUT FILE TO CLOSED
460 RTS
465 ; CLOSE OUTPUT FILE
470 CLOSE2 LDA FOPEN2; CHECK TO SEE IF OUTPUT FILE IS OPEN
480 BEQ CLOSE4; IF NOT EXIT
490 LDA #<CLOSEW
```

```
500 STA FMOP
510 LDA #>CLOSEW
520 STA FMOP+1
530 JSR FMDRVR
540 LDA #0
550 STA FOPEN2; SET OUTPUT FILE TO CLOSED
560 RTS
570 CLOSE4 RTS; CLOSE NOT NEEDED FOR PRINTER
580 FMDRVRO LDY #08; PUT FILENAME INTO PARAMETER FIELD
590 LDA (FMOP),Y
600 STA PARM
610 INY
620 LDA (FMOP),Y
630 STA PARM+1
640 LDA #<FILEN
650 STA TEMP
660 LDA #>FILEN
670 STA TEMP+1
680 LDY #00
690 LDA #$A0
700 PADFN STA (PARM),Y; FIRST FILL WITH SPACES
710 INY
720 CPY #31
730 BNE PADFN
740 LDY #00
750 FMO LDA (TEMP),Y; THEN PUT FILENAME IN PARM
760 ORA #$80; MAKE SURE HIGH BIT SET
770 STA (PARM),Y
780 INY
790 CPY FNAMELEN
800 BNE FMO
```

```
810 FMDRVR JSR $3DC; GET START ADDRESS TO PARAMETER FIELD
820 STA PARM+1
830 STY PARM
840 LDY #00
850 PARMSU LDA (FMOP),Y; PUT PARMS INTO PARM
860 STA (PARM),Y
870 INY
880 CPY #18
890 BNE PARMSU
900 LDX #00
910 JSR $3D6; JSR TO FILE MANAGER IN DOS
920 RTS
925 ; SET CURRENT INPUT CHANNEL
930 CHKIN STX OPNI
940 RTS
945 ; SET CURRENT OUTPUT CHANNEL
950 CHKOUT TXA
960 STA OPNO
970 CPX #4; IF PRINTER THEN
980 BNE CHKOUTO
990 LDA #<PRNTRO; SET OUTPUT TO PRINTER
1000 STA CSWD
1010 LDA #>PRNTRO
1020 STA CSWD+1
1030 CHKOUTO RTS
1035 ; GET ONE BYTE FROM CURRENTLY OPEN CHANNEL
1040 CHARIN STY Y1
1050 STX X; SAVE X & Y REG
1060 LDA OPNI; CHECK TO SEE IF INPUT CHANNEL
1070 CMP #1
1080 BNE CTOUT; IF NOT EXIT
```

```
1090 JSR RDBYTE
1100 PHP
1110 LDY Y1
1120 LDX X
1130 PLP
1140 RTS
1150 CTOUT LDY Y1
1160 RTS
1165 ; OUTPUT ONE BYTE TO CURRENTLY OPEN CHANNEL
1170 PRINT STY Y1; SAVE REG
1180 STA A1
1190 LDA OPNO; CHECK TO SEE IF TO OUTPUT FILE
1200 CMP #02
1210 BNE NXT1
1220 LDA A1; YES, WRITE THE BYTE
1230 JSR WRBYTE
1240 JMP CTOUT
1250 PRNTRO STA A1; PRINTER OUTPUT ROUTINE
1260 CMP #$8D
1270 BNE PROUT
1280 LDA #10
1290 PROUT STA PRNTR
1300 NOTDONE LDA PRNTRDN
1310 BMI NOTDONE
1320 LDA A1
1330 RTS
1340 NXT1 LDA OPNO; CHECK TO SEE IF TO PRINTER
1350 CMP #4
1360 BNE NXT2
1370 LDA A1; YES, PRINT TO PRINTER
1380 JSR PRNTRO
```

```
1390 JMP CTOUT
1400 NXT2 LDA A1; NO, MUST BE TO SCREEN
1410 ORA #$80
1420 JSR COUT
1430 JMP CTOUT
1435 ; CLOSE ALL INPUT AND OUTPUT CHANNELS
1440 CLRCHN LDA #00
1450 STA OPNO
1460 STA OPNI
1470 LDA #$F0; RESET OUTPUT ROUTINE
1480 STA CSWD
1490 LDA #$FD
1500 STA CSWD+1
1510 RTS
1515 ;CHECK FOR STOP KEY
1520 STOPKEY LDA $C000
1530 CMP #$83
1540 RTS
1545 ; CLOSE OPEN FILES
1550 CLOSE CMP #01
1560 BNE CL2; CLOSE INPUT FILE?
1570 JMP CLOSE1
1580 CL2 CMP #02; NO, CLOSE OUTPUT FILE?
1590 BNE CL4
1600 JMP CLOSE2
1610 CL4 JMP CLOSE4; NO, MUST BE PRINTER
1700 ; BASIC WEDGE
1710 WEDGE STA A1
1720 LDA #$00; IS TXTPTR AT $200?
1730 CMP TXTPTR
1740 BNE OUT
```

```
1750 LDA #02
1760 CMP TXTPTR+1
1770 BNE OUT; NO, EXIT
1775 LDY #0
1780 NXTCHR LDA (TXTPTR),Y; IGNORE LEADING SPACES
1781 CMP #32
1782 BNE ISLNUM
1783 INC TXTPTR
1784 JMP NXTCHR
1790 ISLNUM CMP #$2F; IS IT A NUMBER?
1800 BCC OUT; NO, EXIT
1810 CMP #$3A
1820 BCC INSLIN
1830 OUT LDA $200; IS IT "ASM "?
1840 CMP #65
1850 BNE OUT1
1860 LDA $201
1870 CMP #83
1880 BNE OUT1
1890 LDA $202
1900 CMP #77
1910 BNE OUT1
1920 LDA $203
1930 CMP #32
1940 BNE OUT1; NO, EXIT
1950 LDY #0; YES
1960 TFRNAM LDA $204,Y; TRANSFER NAME TO TOP OF SCREEN
1970 CMP #0
1980 BEQ ASM
1990 ORA #$80
2000 STA $400,Y
```

```
2010 INY
2020 JMP TFRNAM
2030 ASM LDA #$A0; PUT FOLLOWING 3 SPACES
2040 STA $400,Y
2050 STA $401,Y
2060 STA $402,Y
2070 PLA; PULL RETURN ADDRESS AND JUMP TO START
2080 PLA
2090 JMP START
2100 OUT1 LDA A1; NORMAL CHRGET
2110 CMP #$3A
2120 BCS EXIT
2130 CMP #$20
2140 BNE NXT
2150 JMP CHRGET
2160 NXT SEC
2170 SBC #$30
2180 SEC
2190 SBC #$D0
2200 EXIT RTS
2210 INSLIN LDX PRGEND; FOUND LINE NUMBER, NOW INSERT LINE
2220 STX VARTAB
2230 LDX PRGEND+1
2240 STX VARTAB+1
2250 CLC
2260 JSR LINGET; GET LINE NUMBER
2270 JSR TOKNIZ
2280 PLA
2290 PLA
2300 JMP LININS; JUMP TO NORMAL INSERT LINE AND RESET LINE LINK ADDRESSES
2310 TOKNIZ LDY #00; TOKENIZE LINE
```

```
2320 STY HIGHDS
2330 LDA #02
2340 STA HIGHDS+1
2350 TK3 LDA (TXTPTR),Y
2360 STA (HIGHDS),Y
2370 INY
2380 CMP #00; END OF LINE
2390 BNE TK3
2400 DEY; YES
2410 TK4 DEY
2420 LDA (HIGHDS),Y; IGNORE FOLLOWING SPACES
2430 CMP #32
2440 BEQ TK4
2450 INY
2460 LDA #0
2470 STA (HIGHDS),Y
2480 INY
2490 INY
2500 INY
2510 INY
2520 INY; Y-REG HOLDS LINE LENGTH +6
2530 RTS
2540 EDITSU LDA #<WEDGE; INITIALIZE WEDGE
2550 STA $BB
2560 LDA #>WEDGE
2570 STA $BC
2580 LDA #$4C; "JMP"
2590 STA $BA
2592 LDA #$FC:STA 115; SET HIMEM
2595 LDA #$79:STA 116
2600 RTS
2610 .FILE FINDMN
```

Program 5-3. Open1, Atari

```
100 OPEN1 JSR CLRCHN
110 LDA #1
120 JSR CLOSE
130 LDA #1
140 STA FNUM
150 LDA #4
160 STA FDEV
170 LDA #0
180 STA FSECOND
190 NAMEAD LDA #<FILEN
200 STA FNAMEPTR
210 LDA #>FILEN
220 STA FNAMEPTR+1
230 JSR OPEN
240 LDA ST
250 BMI OPENERR
260 LDA RAMFLAG
270 BEQ NOLOAD
280 JSR AFTEROPEN
290 LDA #<TEXTBAS
300 STA PMEM
310 LDA #>TEXTBAS
320 STA PMEM+1
330 NOLOAD RTS
340 OPENERR JSR ERRPRINT
350 JMP TOBASIC
360 OPEN2 LDA #2
370 STA FNUM
380 LDA #8
390 STA FDEV
400 LDA #0
410 STA FSECOND
420 LDA #<FILEN
430 STA FNAMEPTR
440 LDA #>FILEN
450 STA FNAMEPTR+1
460 LDA #2
470 JSR CLOSE
480 LDA ST
490 BMI OPENERR
500 JSR OPEN
510 LDX #2
520 JSR CHKOUT
530 LDA #255
540 JSR PRINT
550 JSR PRINT
560 LDA TA
570 JSR PRINT
580 LDA TA+1
590 JSR PRINT
600 LDA LLSA
610 JSR PRINT
```

```
620 LDA LLSA+1
630 JSR PRINT
640 JSR CLRCHN
650 RTS
660 OPEN4 LDA #4
670 STA FNUM
675 JSR CLOSE
680 LDA #8
690 STA FDEV
700 LDA #0
710 STA FSECOND
720 LDA #2
730 STA FNAMELEN
740 LDA #<PNAME
750 STA FNAMEPTR
760 LDA #>PNAME
770 STA FNAMEPTR+1
800 JSR OPEN
810 LDA ST
820 BMI OPENERR
830 JSR CLRCHN
840 RTS
850 PNAME .BYTE 80 58
860 .FILE D:FINDMN.SRC
```

Program 5-4. Findmn

```
10 ; "FINDMN" -- LOOKS THROUGH MNEMONICS FOR MATCH TO LABEL.
20 ; WE JMP TO THIS FROM EVAL.  & JMP BACK TO 1 OF 2 LOCATIONS (JMP FOR SPEED)
30 FINDMN LDY #0
40 LDX #255; PREPARE X TO GO TO ZERO AT START OF LOOP
50 LOOP INX; X RAISED TO ZERO AT START OF LOOP
60 LDA MNEMONICS,Y; LOOK IN TABLE OF MNEMONICS
70 CMP LABEL; COMPARE IT TO 1ST CHAR. OF WORD IN LABEL BUFFER
80 BEQ MORE; IF =, COMPARE 2ND LETTERS OF TABLE VS. BUFFER
90 INY; OTHERWISE GO UP THREE IN THE TABLE TO FIND THE NEXT MNEMONIC
100 INY
110 INY
120 CPX #57; HAVE WE CHECKED ALL 56 MNEMONICS.
130 BNE LOOP; IF NOT, CONTINUE TRYING TO FIND A MATCH
140 NOMATCH JMP EQLABEL; DIDN'T FIND A MATCH (SO GO BACK TO EVAL)
150 MORE INY; COMPARE 2ND LETTER
```

```
160 LDA MNEMONICS,Y
170 CMP LABEL+1
180 BEQ MORE1; IF =, GO ON TO COMPARE 3RD AND FINAL LETTER
190 INY
200 INY
210 BNE LOOP; 2ND LETTER DIDN'T MATCH, TRY NEXT MNEMONIC (Y <> 0)
220 BEQ NOMATCH ; IF Y = 0, WE'VE GONE PAST TABLE (RETURN TO EVAL)
230 MORE1 INY; COMPARE 3RD LETTER
240 LDA MNEMONICS,Y
250 CMP LABEL+2
260 BEQ FOUND; IF 3RD LETTERS ARE =, WE'VE FOUND OUR MATCH
270 INY
280 BNE LOOP; OTHERWISE TRY NEXT MNEMONIC
290 BEQ NOMATCH
300 FOUND LDA LABEL+3; THE 4TH CHAR. MUST BE A BLANK FOR THIS TO BE A MNEMONIC
310 CMP #32
320 BEQ FO1; IF SO, STORE DATA ABOUT THIS MNEMONIC & RETURN TO EVAL.
330 CMP #0; OR IF END OF LINE, IT WOULD BE AN IMPLIED ADDR. MNEMONIC LIKE INY
340 BNE NOMATCH; OTHERWISE, NO MATCH FOUND (IT'S NOT A MNEMONIC).
350 FO1 LDA TYPES,X; STORE ADDR. TYPE.
360 STA TP
370 LDY OPS,X; STORE OPCODE
380 STY OP
390 END JMP EVAR; MATCH FOUND SO JUMP TO EVAR ROUTINE IN EVAL
400 .FILE GETSA
```

For the Atari version of Findmn, change line 400 to:

400 .FILE D:GETSA.SRC

Program 5-5. Getsa

```
10 ; "GETSA" GET STARTING ADDRESS FROM DISK (LEAVES DISK POINTING AT-
20 ;                    *= THIS SPACE (START ADDRESS)
30 ;   (EXPECTS FILE #1 TO BE ALREADY OPENED).
40 ; ------------------------
50 GETSA LDX #1; SET UP INPUT CHANNEL FOR A DEVICE (TO GET BYTES)
60 JSR CHKIN; BASIC'S ROUTINE
70 LDX #6; WE NEED TO THROW AWAY THE 1ST 6 BYTES ON A DISK FILE (LINE LINK,
80 LSA JSR CHARIN; LINE #, AND 2 BYTES) (CHARIN IS "GET BYTE")
90 DEX; COUNT DOWN UNTIL WE'VE PULLED OFF THE
100 BNE LSA; 1ST 6 BYTES...THEN-----------------
110 JSR CHARIN; PULL IN NEXT BYTE
120 CMP #172; IS IT THE * SYMBOL
130 BEQ MSA; IF SO, GO BACK TO CALLER (EVAL SUBPROGRAM CALLS GETSA)
140 LDA #<MNOSTART; OTHERWISE, PRINT ERROR MESSAGE WHICH
150 STA TEMP; SAYS "NO START ADDRESS".  POINT TO THIS ERROR MESSAGE IN
160 LDA #>MNOSTART; THE POINTER,"TEMP," AND PRINT THE MESSAGE (PRNTMESS)
170 STA TEMP+1; (NOTE: THIS NO-START-ADDRESS CONDITION OCCURS 2 WAYS: EITHER
180 JSR PRNTMESS; YOU FORGOT TO WRITE ONE OR YOU GAVE THE WRONG FILE NAME)
190 JMP FIN; GO BACK TO BASIC VIA THE SHUTDOWN ROUTINE WITHIN EVAL;-------
200 MSA RTS
210 .FILE VALDEC
215 ;
217 ; ------------------------
220 ; "MEMSA" GET STARTING ADDRESS FROM MEMORY.  LEAVES DISK POINTING AT-
230 ;                    *= THIS SPACE (START ADDRESS)
240 ; !! INITIALIZES PMEM TO START OF MEMORY
250 ; REPLACES "GETSA" SOURCE CODE FILE TO CREATE RAM-BASED ASSEMBLER.
260 ; ------------------------
270 MEMSA LDA RAMSTART:STA PMEM:LDA RAMSTART+1:STA PMEM+1
```

```
28Ø LDX #3:MEM1 JSR CHARIN:DEX:BNE MEM1; ADD 4 TO PMEM TO POINT TO *=
3ØØ JSR CHARIN:CMP #172:BEQ MMSA
31Ø LDA #<MNOSTART:STA TEMP:LDA #>MNOSTART:STA TEMP+1:JSR PRNTMESS
32Ø JMP FIN; GO BACK TO BASIC VIA ROUTINE WITHIN EVAL
33Ø MMSA RTS
34Ø ; --------------------------
35Ø ; "NEW CHARIN"  ASSEMBLE SOURCECODE FROM MEMORY RATHER THAN DISK.
36Ø ;  (IMITATES CHARIN FOR DISK)
37Ø ;  RETURNS WITH NEXT BYTE FROM MEMORY, IN A
38Ø ; --------------------------
39Ø CHARIN INC PMEM:BNE INCP1:INC PMEM+1; REPLACES CONVENTIONAL CHARIN/DISK
4ØØ INCP1 STY Y:LDY #Ø:LDA (PMEM),Y:PHP:LDY Y:PLP:RTS; SAVE STATUS REGISTER
41Ø CHKIN RTS; REPLACES DISK ROUTINE IN DEFS
42Ø ; --------------------------
43Ø ;
44Ø ;        .. THE OTHER NECESSARY MODIFICATIONS ..
45Ø ;
46Ø ;HERE ARE THE REST OF THE MODIFICATIONS WHICH CHANGE LADS FROM
47Ø ;DISK-BASED TO RAM-MEMORY-BASED SOURCE CODE ASSEMBLY:
48Ø ;
49Ø ;
5ØØ ;  1. REMOVE DEFINITIONS OF CHARIN AND CHKIN (IN THE DEFS FILE)
51Ø ;        (JUST INSERT A SEMICOLON AS THE 1ST CHARACTER
52Ø ;         IN LINES 22Ø AND 24Ø OF THE DEFS SOURCE CODE FILE.)
53Ø ;
54Ø ;  2. REPLACE "JSR GETSA" IN LINE 37Ø OF THE EVAL FILE WITH
55Ø ;          "JSR MEMSA" AND REMOVE THE "JSR OPEN1" IN LINE 35Ø AND
56Ø ;           LINE 435Ø IN EVAL.
57Ø ;
58Ø ;  3. PUT A ; IN FRONT OF ".FILE VALDEC" IN LINE 21Ø IN THIS FILE.
59Ø ;         (IN OTHER WORDS, ALLOW THE NEW VERSIONS OF CHARIN ETC.
```

```
600 ;             TO ASSEMBLE INTO THE FINISHED VERSION OF LADS.)
610 ;
620 ;   4. PUT SEMICOLONS AS 1ST CHARACTER IN LINES 760,770,780, & 800
630 ;            IN THE PSEUDO SUBPROGRAM.  ALSO, CHANGE LINE 750 TO
640 ;            READ "JSR LOAD1" (INSTEAD OF "JSR OPEN1").
650 ;
660 .FILE VALDEC
```

Program 5-6. Getsa, Apple Modifications

To create the Apple version of Getsa, make the following changes and additions to Program 5-5:

```
75 LSA STX X
80 JSR CHARIN; RAM START ADDRESS, AND LINE LINK) (CHARINIS "GET BYTE")
85 LDX X
120 CMP #$2A; IS IT THE * SYMBOL
```

Program 5-7. Getsa, Atari Modifications

To create the Atari version of Getsa, omit lines 215–660 in Program 5-5 and change the following lines:

```
10 ;ATARI MODIFICATIONS--GETSA
50 GETSA LDA #<TEXTBAS:STA PMEM:LDA #>TEXTBA
   S:STA PMEM+1
55 LDX #1
70 JSR LINENUMBER
80 ;
```

```
90 ;
100 ;
120 CMP #42
210 .FILE D:VALDEC.SRC
```

Program 5-8. Valdec

```
10 ; "VALDEC"  TRANSLATE ASCII INPUT TO A TWO-BYTE INTEGER IN RESULT
15 ;
20 ;  SETUP/TEMP MUST POINT TO ASCII NUMBER (WHICH ENDS IN ZERO).
30 ;  RESULTS/ RESULT HOLDS 2-BYTE RESULT
40 ; --------------------------
50 VALDEC LDY #0
55 ; READ ASCII FROM LEFT TO RIGHT--INCREMENTING Y --(TO FIND LENGTH)
60 VGETZERO LDA (TEMP),Y
70 BEQ VZERO; 0 DELIMITER FOUND
80 INY
90 JMP VGETZERO;---------------  (FOR EXAMPLE, ASSUME ASCII IS "15")
110 VZERO STY VREND; SAVE LENGTH OF ASCII NUMBER  (IN THE EXAMPLE, LEN = 2)
120 DEY
130 LDA #0; CLEAN "RESULT" VARIABLE (SET TO 0)
140 STA RESULT
150 STA RESULT+1
160 LDX #1; USE "X" VARIABLE AS A MULTIPLY-X10-HOW-MANY-TIMES COUNTER
170 STX X
180 VALLOOP LDA (TEMP),Y; LOAD IN THE RIGHTMOST ASCII CHARACTER (EX: "5")
190 AND #$0F; AS ASCII, 5 = $35.  O STRIP OFF THE 3, LEAVING THE 5.
200 STA RADD; STORE IN MULTIPLICATION REGISTER
210 STA TSTORE; STORE IN "REMEMBER IT" REGISTER
220 LDA #0; PUT 0 IN BOTH THESE REGISTERS (IN THEIR HIGH BYTES)
```

```
230 STA RADD+1
240 STA TSTORE+1;---------------- MULTIPLY X10 AS MUCH AS NECESSARY------
250 VLOOP DEX; LOWER THE COUNTER. (IN THE EXAMPLE, X NOW = 0 FOR 1ST CHAR)
260 BEQ VGOON; SO WE DON'T JSR TO THE X10 SUBROUTINE IN THIS CASE)
270 JSR TEN; OTHERWISE,WE'D MULTIPLY THE NUMBER X10 AS MANY TIMES AS NECESSARY
280 LDA RADD; MOVE RESULT OF MULTIPLICATION INTO STORAGE REGISTER
290 STA TSTORE
300 LDA RADD+1
310 STA TSTORE+1; SAVING RESULTS OF MOST RECENT MULTIPLICATION
320 JMP VLOOP; CONTINUE MULTIPLYING X10 UNTIL X IS DOWN TO ZERO.--------
330 VGOON INC X; RAISE X BY 1 (SINCE WE'RE MOVING LEFT AND EACH NUMBER WILL
335 ;                              BE 10X THE ONE TO ITS RIGHT).
340 LDX X
350 JSR VALADD; ADD RADD TO RESULT (ADD IN RESULTS OF THE MULTIPLICATION)
360 DEY; MOVE INDEX OVER BY 1 (TO POINT TO NEXT ASCII CHAR. TO THE LEFT)
370 DEC VREND; LOWER LENGTH POINTER.  IF IT'S NOT YET ZERO, THEN
380 BNE VALLOOP; CONTINUE PROCESSING THIS ASCII NUMBER
390 RTS; OTHERWISE RETURN TO CALLER.
400 ;---------------   MULTIPLY BY 10
410 TEN CLC
420 ASL RADD;       MULTIPLY RADD X 4
430 ROL RADD+1
440 ASL RADD
450 ROL RADD+1;---------------
460 CLC
470 LDA TSTORE;PULL OUT ORIGINAL NUMBER AND ADD IT TO RESULT OF X4 (GIVING X5)
480 ADC RADD
490 STA RADD
500 LDA TSTORE+1
510 ADC RADD+1
520 STA RADD+1;------------ NOW, MULTIPLY X2.  ((N*4+N)*2) IS N*10
```

```
530 ASL RADD
540 ROL RADD+1
550 RTS
560 ;------------- ADD RESULTS OF THE MULTIPLICATION TO THE INTEGER ANSWER
570 VALADD CLC
580 LDA RADD
590 ADC RESULT
600 STA RESULT
610 LDA RADD+1
620 ADC RESULT+1
630 STA RESULT+1
640 RTS
650 .FILE INDISK
```

Program 5-9. Valdec, Atari Modifications

To create the Atari version of Valdec, make the following changes and additions to Program 5-8:

```
10 ;ATARI MODIFICATIONS--VALDEC
61 CMP #48
62 BCC VZERO
63 CMP #58
70 BCS VZERO
650 .FILE D:INDISK.SRC
```

Chapter 6

Indisk:
The Main Input Routine

Indisk:
The Main Input Routine

It's up to the Indisk subprogram to pull in a logical line of source code and set it up so that Eval can evaluate it. What does the word *logical* mean when used this way? You'll sometimes hear of a "logical" string or a "logical" line versus a "physical" string or line. The logical thing is what the computer will see and compute. The physical thing might well be longer or shorter.

For example, on the Apple, Atari, and Commodore 64, the screen permits a *physical* line of only 40 characters. And though each screen line can hold only 40 characters, Commodore BASIC can interpret 80-character lines, Apple can interpret 256-character lines, and the Atari can interpret 120-character lines. The *logical* line length is 80, 256, or 120 characters, but the *physical* line is 40. To describe Indisk's routines, we'll need to make a similar distinction.

Two physical lines of LADS source code might be:

```
100 LDA 15: INY:RTS
110 DEC 15
```

but there are four logical lines in these two physical lines:

```
LDA 15
INY
RTS
DEC 15
```

Put another way, the LADS logical line is sometimes smaller than its physical line. The logical item is the piece that a computer—or in this case, LADS—will work with. Whenever you see a colon, you're at the end of a logical line.

In addition to setting up each logical line for examination by Eval, Indisk also performs some other tasks. It sets flags up in response to several pseudo-ops; it transforms single-byte tokenized BASIC keywords into ASCII words (? becomes PRINT); it transforms ASCII hex numbers like $1500 into two-byte integers (the same thing the Valdec subprogram does for ASCII *decimal* numbers); and it handles the important .BYTE pseudo-op. Indisk is a busy place. It's the second longest source file in LADS. Eval interprets logical lines of source code; Indisk prepares them for that interpretation.

Total Buffer Cleaning

Indisk starts by cleaning out an entire group of buffers: LABEL, BUFFER, BUFM, HEXBUF, FILEN, NUBUFF. That's easy because they are all stuck together (see lines 290–340 in the Tables subprogram). The CLEANLAB subroutine in Eval just sticks 0 into the entire string of buffers.

Then 0 is put into the HEXFLAG (is it a $ type number?), BYTFLAG (is it a < or > pseudo-op?), and PLUSFLAG (is it a + pseudo-op?). These three flags will later be set up, if necessary, by Indisk. We want them down, however, at the start of our analysis of each logical line.

At line 110 LADS sees if the previous logical line ended in a colon. LADS tries to be forgiving. It knows that the programmer might accidentally write source code like:

```
100 LDA 15:     LDX 12
```

leaving some spaces between a colon and the start of the next logical line. Rather than crash trying to find a label called blank-blank-L-D-X, it ignores leading blanks following colons. Elsewhere, LADS ignores blanks preceding semicolons. This gives the user complete freedom to ignore that potential punctuation problem. Logical lines with extra blank spaces will be correctly analyzed.

If a colon ended the previous logical line, we need to skip over the fetch-and-store-line-number routine (**130–160**) since there is a line number only at the start of a physical line. In BASIC programs, and consequently in LADS source code, the two bytes just preceding the start of the code proper in each physical line are the line number. They need to be remembered by LADS for printouts and also for error reporting.

The Suction Routine

Lines 170–190 are the suction routine for blanks which might precede a colon. We just loop here until something other than the blank character (#32) is encountered. Notice that this loop is also performed at the start of a physical line, but will have no effect since the computer removes any leading spaces when you first type in a BASIC or LADS line.

Line 210 is the start of the main loop which pulls in each character from the disk, one at a time. We skip over this (**200**) if we've entered at Indisk and therefore are starting a line rather than just looking at the next character *within* a line.

But let's assume for now that we're trying to get the next character in a line. If it's zero, that means the end of a physical line (**230**), so we go to the routine which checks to see if we're at the end of the entire program, not just the end of a single line.

If there was no zero, we check for a colon and jump to the routine which handles that (**260**). Then we check for a semicolon. The next section (**290–750**) handles semicolons. There are two types of semicolon situations, requiring two different responses.

One type of semicolon defines an entire line as a comment. The semicolon, announcing that a remark follows, appears in this case as the first character in a physical line:

100; THIS ENTIRE LINE IS A REMARK.

This type is relatively simple since there is no source code for Eval to evaluate.

The other type of remark, though, appears at the end of a logical line, and there *is* something for Eval to assemble on such lines:

100 LDA 75; ONLY PART OF THIS LINE IS A REMARK.

When we first detect a semicolon (**270**), we store the Y Register in variable A (**290**). The Y Register is very important in Indisk. It is set to zero at the start of each physical line (**60**) and *will still be zero* in line 290 if the semicolon is the first character in a physical line. This is how we can tell which type of comment we're dealing with (at the start of a line or within a line).

If, however, the programmer has not requested a screen printout, there is no point to storing a comment. Comments have no meaning to the assembler; they're just a convenience to the programmer. Line 300 checks to see if PRINTFLAG is set and, if not, skips over the store-the-comment routine.

BABFLAG for Comments

But if the PRINTFLAG was up (contained a 1), we transfer that 1 to force the BABFLAG up as well. BABFLAG tells LADS that there's a comment to be printed after the source and object codes have been printed to screen or printer.

Then that previously stored Y Register is pulled back out, and we see which kind of comment we're dealing with. If Y isn't zero, we've got a within-the-line comment, and we can

JSR to the PULLREST subroutine which stores comments in the comment buffer (**350**). Then we return to Eval to assemble the first part of the line, the source code part (**360**).

When a semicolon appears at the start of a line, though, we'll just fill LABEL, the main buffer, with the comment and then print out that kind of line right here within Indisk. (Printouts are normally controlled by Eval following the assembly of source code. This type of line, however, contains no source code.)

A little loop (**370–440**) stuffs the comment line into LABEL. It exits when it finds the end of a physical line (**380**), and it JSRs when it comes upon a tokenized keyword like PRINT or STOPIT. (STOPIT would appear as three characters in the source code: the token for BASIC's STOP command, and the letters I and T.) Tokenized words have to be stretched out to their ASCII form, or the comment could contain strange nonprinting characters or graphics characters, etc., when printed out. Any character larger than 127 is not a normal alphabetic character. It's going to be a token.

When we finally come upon the end of this physical comment line, we land at PUX1 (**450**) and proceed to print the line number, the comment, and a carriage return just as we do for any other line. Then we put 0 into the A variable to let MPULL (the return-to-Eval subroutine) know that there is no source code to assemble in this line. It will send us back to two different places in Eval, depending on whether we should or shouldn't try to assemble the line currently held in the LABEL buffer.

Storage to BABUF

The PULLREST routine (**520–600**) is similar to the PUX routine above it, but it stores a comment into the BABUF buffer. PULLREST cannot use the LABEL buffer because this is one of those lines where the comment comes *after* some legitimate source code. And Eval assembles all legitimate source code from the LABEL buffer. After Indisk turns the following line over to Eval:

```
100 LDX 22; HERE IS A COMMENT.
```

the two buffers hold their respective pieces of this line:

```
LABEL    LDX 22
BABUF     HERE IS A COMMENT.
```

BABFLAG is set up to alert Eval to print a comment after it has assembled and printed out the LDX 22 part of this line (**520**). Then the semicolon in the Accumulator is saved in the A Register. This is our end-of-line condition. Logical lines can also end with colons and zeros. Different end-of-line conditions require different kinds of exits from Indisk. For example, if we hit a colon, we shouldn't pull in the next two characters and store them as a line number. A colon means we've not yet reached the end of the *physical* line. Since PULLREST is used as a subroutine in various ways—JSRed to from various places in Indisk—it must save the end-of-line condition.

KEYWAD

Then PULLREST pulls the rest of the line into BABUF (**560–650**) with a little detour to KEYWAD if the seventh bit is set on one of the characters being pulled in. That signals a tokenized keyword like ? for PRINT. KEYWAD is the same routine as KEYWORD (called above when Indisk is pulling in source code characters). The only difference between them is that KEYWORD extends ? to the word *PRINT* in LABEL, the source code buffer. KEYWAD extends tokens into BABUF, the comment buffer.

PULLRX (**660–680**) is quite similar to PULLREST. However, PULLRX is a pure suction routine. It pulls in the rest of a comment line, but doesn't store any of the characters. It is called upon when the PRINTFLAG is down and nothing needs to be printed to screen or printer. All PULLRX does is get us past the comment to the next physical line.

MPULL (**690–750**) is the exit from Indisk back to Eval after a commented line has been handled. Recall that there are two kinds of comments—those which take up an entire physical line and those which take up only the latter part of a line, those which come after some real source code. MPULL distinguishes between them after first checking to see if we're at the end of the entire program (ENDPRO). It loads in the A variable. If A is holding a zero, that would mean that the semicolon was the first character in the physical line, and consequently, the entire line was a comment and can be ignored. There's nothing to assemble. So we PLA PLA to get rid of the RTS address and JMP directly to STARTLINE in Eval to get a new physical line.

Y Is the Pointer

Alternatively, if the semicolon was not at the start of the line, the value in the A variable will be higher than zero. (The Y Register was stored in A when a semicolon was first detected [**290**].) Y keeps track of which position we are currently looking at within each physical line. In cases where there *is* some source code on a line for Eval to assemble, we just RTS (**750**) back to Eval where the evaluation routine begins.

The end of the main Indisk loop is between lines 760 and 950. This section is an extension of the character-testing sequence found between lines 220 and 270. What's happening is that a single character is being drawn in from the source code (on a disk file or within RAM memory, depending on which version of LADS you are using). Each character is tested for a variety of conditions: pseudo-ops, keyword tokenization, hex numbers, end-of-line (**220**), colon (**240**), and semicolon (**270**). If it was a semicolon, we dealt with it before making any further tests. The semicolon (comments) handler is the large section of code we just discussed (between lines 290 and 750). If the character isn't a semicolon, however, there are several other special cases which we should test for before storing the character into LABEL, the source code buffer.

Special Cases

Is it a > pseudo-op? If so, we go to the routine which handles that (**770**) called HI. Is it the < pseudo-op? Then go to the LO routine. Is it the plus sign, signaling the + pseudo-op? If not, jump over line 820. The + pseudo-op is handled elsewhere in LADS; all we do for now is set up the PLUSFLAG (**820**). Is it the *=, the Program Counter changing pseudo-op? If so, go to the subroutine which fixes that (**850**). Is it one of the pseudo-ops which start with a period, like .BYTE or .FILE? If so, go to the springboard to the subroutines which deal with these various pseudo-ops (**870**). Is the character a $, meaning that the source code number which follows the $ should be translated as a hex number? If so, go to the hex number routine springboard (**890**).

The final test is for tokenized keywords (? for PRINT). Tokens all have a value higher than 127, so their seventh bit will be set. If the character is lower (BCC) than 127, we can finally add the character to the source code line we're building in the LABEL buffer (**930**). Then we raise the Y Register to point to the next available space in the LABEL buffer, and return to fetch the next available space in the LABEL buffer, and return to fetch the next character of source code from disk or RAM memory (**950**).

This ends the main loop of the Indisk routine. As you see, there are many tests before a character can be placed into the LABEL buffer. We only want to give Eval source code that it can assemble. We can't give it characters like . or + or $ which it cannot evaluate properly. Those, and other special conditions, are worked out and fixed up by Indisk before LADS turns control back to the Eval subprogram.

The Colon Logical End-Of-Line

One special condition is the colon. It is handled at the very start of Indisk as a new physical line is analyzed (**110**). Not much needs to be done with colons except to ignore them. But we do need to prevent LADS from trying to locate the next physical line number. Colons signify the end of a logical line, not the end of a physical line. COLFLAG tells Indisk not to look for a line number. COLFLAG is set whenever a colon is detected (**260**). We jump down to COLON (**970**) and set the flag. We don't need to LDA #1:STA COLFLAG because we wouldn't be here unless the Accumulator was holding a colon character (it's higher than 0). We can just stuff that character into COLFLAG. As long as a flag isn't holding a 0, it's set. When setting flags, it doesn't matter that the number in the flag is higher than 1. Just so it's not 0.

There are two springboards at 990–1020. Recall that branch instructions like BNE cannot go further than 128 bytes in either direction, so you'll get a BRANCH TOO FAR error message from LADS from time to time when you exceed this limit. In such cases, just BNE SPRINGBOARD; just branch to a line you insert, like 990, which just has a JMP to your true target.

Like the . pseudo-op interpreter subroutine, the hex translator is also too far from the branch which tries to reach it. With a hex number, though, we first put the $ into the LABEL buffer so it will be printed when the source code line is sent to the screen or printer. Then we bounce off to the hex translator subroutine (**1020**).

KEYWORD (**1040–1210**) translates one of BASIC's tokens into a proper English word. A BASIC word like PRINT is a *word* to us programmers, but an *action*, a *command*, to the computer. To save space, many versions of BASIC translate the words into a kind of code called "tokens." The token for PRINT might be the number 153, which can fit into a single byte. The *word* PRINT takes up five bytes.

But BASIC itself must detokenize when it lists a program. It must turn that 153 back into the characters P-R-I-N-T. To do that, it keeps a table of the keywords in ROM. We'll take advantage of that table to do our own detokenization.

The specifics of the example we'll examine here are for Commodore computers. The principle, however, applies to Apple and Atari as well. Only the particular numbers differ. We arrive here at KEYWORD because we picked up a character with a value higher than 127. The first thing we do is subtract 127. That will give us the *position* of this keyword in the table of keywords. To see how this works, look at how these words are stored in ROM memory:

enDfoRnexTdatA

Notice that BASIC stores words in this table with their last letter shifted, similar to the way LADS stores labels with their first letter shifted. That's how the start of each word can be detected. The code for these words is set up so that END = 128, FOR = 129, NEXT = 130, and so on.

Imagine that we picked up a 129 and came here to the KEYWORD subroutine to get the ASCII form of the word, the readable form. We would subtract: 129 − 127 = 2. Then we would look for the second word in the table. We store the results of our subtraction in the variable KEYNUM (**1060**) and keep DECing KEYNUM until it's zero and we've thus located the word. We look at the first character in the table of keywords. It will be an *e*. If it's not a shifted character, we've not yet come to the end of a word, and we keep looking (**1120**). Otherwise, we go back and DEC KEYNUM. All of this is just a way of counting through the keyword table until we get to the word we're after.

When we find it (**1140**), we store the ASCII characters from the table into LABEL, our main input buffer. Again, a shifted character in the table shows us that we've reached the *end* of the word (**1160**), and we can return to the *caller* (the routine we JSRed here from) after clearing out the seventh bit.

KEYWORD turns this line (in the source code):

100 START? LDA [IT (two embedded keyword tokens, ? and [)

into:

100 STARTPRINT LDA RUNIT (which we can read from screen or printer)

The HI subroutine (**1230**) handles the > pseudo-op which gets the high byte of a two-byte label as shown in Listing 6-1.

Listing 6-1

```
 5Ø SCREENPOINTER = $FD;  ZERO PAGE POINTER FOR INDIRECT Y ADDRESSING
1ØØ SCREEN = $Ø4ØØ;  DEFINE START OF SCREEN RAM
11Ø LDA #>SCREEN; LOAD IN HIGH BYTE OF SCREEN ADDRESS
12Ø STA SCREENPOINTER+1;  STORE IT IN HIGH BYTE OF SCREEN POINTER
13Ø LDA #<SCREEN;  LOAD IN LOW BYTE OF SCREEN ADDRESS
14Ø STA SCREENPOINTER;  STORE IT IN LOW BYTE OF SCREEN POINTER
```

Listing 6-2

```
1Ø *= 8ØØ
1ØØ LDA 15
11Ø JMP CONTINUE; (AT THIS POINT WE'RE AT ADDRESS 8Ø5)
12Ø *= 855 (THIS RESETS THE PC TO 855)
13Ø CONTINUE INY; (THIS WILL ASSEMBLE AT ADDRESS 855,
14Ø ;              LEAVING A 5Ø-BYTE-LONG BUFFER OR
15Ø ;              STORAGE ZONE FOR VARIABLES.)
```

Listing 6-3

```
1Ø  32Ø *= 8ØØ
1ØØ 32Ø A5 ØF       LDA 15
11Ø 322 4C 58 Ø3    JMP CONTINUE; (AT THIS POINT WE'RE AT ADDRESS 8Ø5)
12Ø 325 *= 855      (THIS RESETS THE PC TO 855)
13Ø 357 C8          CONTINUE INY; (THIS WILL ASSEMBLE AT ADDRESS 855,
14Ø;                LEAVING A 5Ø-BYTE-LONG BUFFER OR
15Ø;                STORAGE ZONE FOR VARIABLES.)
```

This sort of thing is fairly common during the initialization phase of an ML program. It prepares for the useful Indirect Y addressing mode (sometimes called Indirect Indexed addressing: LDA (LABEL),Y). The > and < pseudo-ops make it easy to set up the zero page pointers upon which Indirect Y addressing depends.

The adjustments necessary to make these pseudo-ops work are performed in the Equate subprogram. All we do here is set up the BYTFLAG to show which of them was encountered. BYTFLAG is 0 normally, set to 1 for a < low byte request and 2 for a > high byte request. Then we go back to fetch the next character in the source code. The > and < symbols are not stored in the LABEL buffer.

Don't Drive with Your Legs Crossed

The STAR subroutine (**1300**) deals with the pseudo-op which changes the Program Counter. This pseudo-op has one primary use: It creates a stable place for tables. Some people like to use it to make room for tables *within* source code (and consequently within the resulting object code too). That seems both unnecessary and dangerous, like driving with your legs crossed. Most of the time it won't do any damage, but when it does cause problems, it causes a crash.

If you like to live dangerously, go ahead and stick a table or a buffer right in the middle of your code. The *= pseudo-op allows coding as shown in Listing 6-2. When assembled, that risky trick will look like the listing shown in Listing 6-3. This example leaves—between $325 and $357—a 50-byte-long zone to be used for data rather than instructions. You must jump over the table. But what's the point? Why not do the sensible thing and put all your tables, register, buffer, etc.—all your nonprogram stuff—in one place? At the end of the entire program. Not only does that ease your programming task by making it simple to understand what you're trying to do, it also allows the *= pseudo-op to make its true contribution to assembling: a stable table.

When you're assembling a long program, you will often go through a two-step process. You'll assemble, then test. The test fails. You change the source code and try it again. This assemble-test rhythm takes place so often that you'll want to make it as easy on yourself as possible. One of your best debugging techniques will involve running your code and then looking in the buffers, registers, variables, and other temporary storage places to see just exactly what is there. That's usually the best clue to what went wrong. If you are trying to load in the word TEXTFILE from disk and your buffer holds EXTFILE0, that tells you exactly what you need to do to fix up the source code.

In other words, you want to be able to check buffers, variables, etc., often. Where are they located in the object code? Obviously, each time you make a slight change to the source code, everything in the object code above the change in memory shifts. All the addresses beyond the changed source code will go up or down depending on whether you added or subtracted something.

Stabilizing Buffers

This makes for very unstable addresses. You would never know where to PEEK at a particular buffer or variable.

There are two ways to solve this. You could put the data buffers, etc., at the start of your program. That way, they wouldn't shift when you changed the source code beyond them. But that's somewhat clumsy. That means that your program doesn't start with the first byte. The entry to your program is up higher, and you can't just SYS or CALL or USR to the first byte.

An alternative, and likely the best, idea is to put tables at the very end. That way the SYS to the object code start address is also the first byte of the ML program. But how does this solve the shifting tables problem? That's where the *= comes in.

When I first started to write LADS, I decided to start it at $3A00. That left plenty of room below for BASIC-type source files and plenty of room above for "Micromon," an extended debugging monitor program which sits in memory between $5B00 and $7000. (I do all my programming on the venerable, but serviceable, Commodore PET 8032.) LADS was expected to end up using about 4K of memory, so I forced Tables, the final source file, to detach itself from the rest of the program and to assemble at $5000. The Tables subprogram started off like this:

```
10; TABLES
20 *= $5000
30 MNEMONICS etc.
```

This kept everything in the Tables unaffected by any changes in the program code below it. The entire source code could be massaged and manipulated without moving the data tables one byte up or down in memory. A detached table is a stable table.

So, during the weeks while LADS was taking shape, I learned the addresses of important buffers like LABEL and important variables and flags. That makes debugging much faster. Sometimes, I could tell what was wrong by simply PEEKing a single flag after a trial run of the source code.

A program the size of LADS, a complex game, or any other large ML program, will require perhaps hundreds of assemblies. It becomes very useful to have learned the special addresses, like buffers, where the results of a trial run of your object code are revealed. And for this reason, these buffer and flag addresses should stay the same from the day you start programming until the day the entire program is composed.

How is the *= pseudo-op handled? Before anything else, we pull in the rest of the source code line by a JSR to STINDISK, the main loop in Indisk. After that, STAR checks to see if anything should be printed out by looking at PASS. On pass 1, we'll skip over the printout (**1320**). Otherwise, we print the star and the input line held in the LABEL buffer. We won't check to see if a printout is requested by looking at PRINTFLAG or SFLAG (screen printout). *= is such a radical event that it will be displayed on pass 2 whether or not any printouts were requested.

Then we come to the familiar hex or decimal number question. Hex numbers are translated and put into the RESULT variable as they stream in. Indisk does hex. Decimal ASCII isn't automatically put into RESULT. If the argument following *= was hex, we skip over the next few lines (**1380**). If not, we look for the blank character (in *= 500, the character between the = and the 5). Finding that (**1420**), we point the TEMP variable to the ASCII decimal number and JSR VALDEC to give the correct value to RESULT. We'll use RESULT to adjust the PC as requested.

Padding the Disk File

If the programmer wants object code stored to disk, we cannot just change the internal LADS program counter. The disk drive won't notice that. We've got to pad the disk program: We've got to physically send spacer bytes to the disk to move its pointer the correct number of bytes forward. Object code is stored only on pass 2.

Thus, two questions are asked here. Does the programmer want object code stored? And is the disk drive a recipient of that object code? If the answer to both questions is "yes," we JSR FILLDISK (**1590**), a padding routine we'll come to later. If not, the whole issue of disk padding doesn't matter and we can proceed to adjust the PC (SA is the variable name of the LADS Program Counter) by transferring RESULT into it (**1600–1630**). Then we PLA PLA the RTS off the stack and jump back into Eval to get the next physical line.

ENDPRO is a short but essential routine. After each physical line we need to see if we've reached the end of the source code program. Microsoft BASIC signals the end of a BASIC program with three zeros.

But before checking for those telltale zeros, ENDPRO fills the buffers with zeros to clean them (**1680–1710**).

Then it pulls in the next two characters. If the second one is a zero, we know it's the end of a source file (not necessarily the end of a series of chained source files; that's flagged by the .END pseudo-op). However, if it is the end of a program file, we flip the ENDFLAG up to warn Eval and RTS back to Eval (**1790**). Even though Indisk has discovered that we're at the same last line in a file, Eval still has that last line to evaluate and assemble. The ENDFLAG won't have any immediate effect when we first return to Eval.

The other possibility is that we won't find the three zeros and that this isn't the last line of a file. If it isn't, we just set the COLFLAG down because at least we're at the end of a physical line. A zero always means that. Then we return to Eval. Indisk just pulls in one line at a time.

Hex Conversions

HEX is an interesting routine. It is called when Indisk detects the $ character. HEX looks at the ASCII form of a number like $0F and turns it into the equivalent two-byte integer 00 0F in RESULT. It's similar to the subprogram Valdec which translates an ASCII decimal number into an integer.

HEX operates like a little Indisk. It pulls in characters from the source code, storing them in its own special buffer, HEXBUF, until it finds either a zero, a colon, a blank, a semicolon, a comma, or a close parenthesis character. Each of these symbols means that we've reached the end of the hex number. Some of them signal the end of a line, some of them don't. Whichever category they fall into, they go to the appropriate routine, DECI or DECIT.

Busy X and Y

If we're not yet at the end of the hex number, however, the character is stored in HEXBUF (**1970**) for later translation and also stored in LABEL for printout. Notice that both the X and the Y Registers are kept busy here, indexing their respective buffers. Y cannot do double duty because it is farther into the LABEL buffer than X; the LABEL buffer is holding the entire logical line, HEXBUF is holding only the ASCII number. The two buffers will look like this when the source line HERE LDA $45 is completely stored:

LABEL　HERE LDA $45
HEXBUF　45

LABEL will be analyzed and assembled by Eval. It needs to store the entire logical line. HEXBUF will be analyzed only to extract the integer value of the hex number. Storing anything else in HEXBUF would be confusing.

A hex number which is not at the end of a line goes to DECIT (**2020**) and, the length of the hex number is stored into the variable HEXLEN (**2020**) so we'll know how many ASCII characters there are to translate into an integer. Then the final character (a comma or whatever) is put into the LABEL buffer. Then the JSR to STARTHEX (**2050**) translates the ASCII into an integer in RESULT. A JMP (rather than a JSR) to STINDISK pulls in the rest of the logical line and takes us away from this area of the code. The assembler will not return to this area. It will treat the rest of the line as if it were an ordinary line.

By contrast, a hex number which is at the end of a line goes to DECI (**2070**), and we store the type of end-of-line condition (colon, semicolon, 0) in the variable A. We put the length of the hex number into the variable HEXLEN (**2090**), so we'll know how many ASCII characters there are to translate into an integer. And we put a 0 delimiter at the end of the information in the LABEL buffer. Then the JSR to STARTHEX (**2110**) translates the ASCII into an integer in RESULT. We restore the colon or semicolon or whatever (**2120**) and jump to the routine which provides a graceful exit (**2130**).

ASL/ROL Massage

STARTHEX turns a hex number from its ASCII form into a two-byte integer. It does this by rolling the bits to the left, pulling the number into RESULT's two bytes, and adjusting for alphabetic hex digits (A–F) as necessary.

The variable HEXLEN knows how many characters are in the hex number. It will tell us how many times to go through this loop. Before entering the loop, we clean the RESULT variable by storing zeros into it (**2140–2160**) and set the X Register to zero.

The loop proper is between lines 2180 and 2350, and is largely an ASL/ROL massage. Each bit in a two-byte number is marched to the left. ASL does the low byte, ROL the high byte. ASL moves the seventh bit of RESULT into the carry. ROL puts the carry into the zeroth bit of RESULT+1, the high byte.

As an example of how this ASCII-to-integer machinery works, let's assume that the number $2F is sitting in the HEXBUF. As ASCII, it would be 2F. But recall that the ASCII code simplifies our job somewhat since the number 2 is coded as $32. To turn an ASCII hex digit into a correct integer, we can get rid of the unneeded 3 by using AND #$0F.

Alphabetic Numbers

What complicates matters, however, is those alphabetic digits in hex numbers: A through F. For them, we'll need to subtract 7 to adjust them to the proper integer value. They, too, will have the high four bits stripped off by AND #$0F.

Let's now follow $2F as it rolls into RESULT. $2F, as two ASCII digits in HEXBUF, is: $32 $46 or, in binary form, 00110010 01000110.

HXLOOP starts off by moving all the zeros in RESULT four places to the left. There are four ASL/ROL pairs. The first time through this loop, just zeros move and there's no effect. Then we load in the leftmost byte from the HEXBUF (**2260**) and see if it's an alphabetic digit. This time we're loading in the $32 (the ASCII 2), so it isn't alphabetic and we branch (to 2300) for the AND which strips off the four high bits:

```
    00110010  ($32, as ASCII code digit)
AND 00001111  ($0F)
    00000010  (now a true integer 2)
```

The ORA command sets a bit in the result if *either* of the tested bits is set. That's one way of stuffing a new value into RESULT:

```
    00000000  (RESULT is all zeros at this point)
ORA 00000010(we're stuffing the integer 2 into it)
    00000010  (leaving an integer 2 in RESULT)
```

Next the X index is raised and compared to the length of the ASCII hex number (in our example $2F, HEXLEN will hold a 2). X goes from 0 to 1 at this point and doesn't yet equal HEXLEN, so we branch back up (**2350**) to the start of the loop and roll the 2 into RESULT, making room for the next ASCII digit:

Carry bit	*high byte*	*low byte*	
0	00000000	00000010	(our 2 before first ASL/ROL)
0	00000000	00000100	(after)
0	00000000	00001000	(after the 2nd ASL/ROL)
0	00000000	00010000	(after the 3rd ASL/ROL)
0	00000000	00100000	(after the 4th and final ASL/ ROL)

What's happened here is that we've shoved the 2 from the low four bits into the high four bits of RESULT. This makes 2 (decimal) into 32 (decimal), or $20. Why do that? Why make room for the next digit in this way? Because the 2 in $2F is really a hex $20. It's a *digit* 2, but not *number* 2. It's *not* a number 2 any more than the 5 in 50 is a 5. This ASL/ROL adjusts each digit to reflect its *position,* and position determines the numeric value of any digit.

Alphabetic Adjustment

Now it's time to pick up the F from HEXBUF (**2260**), and since it has a decimal value of 70, it *is* higher than 65, so we adjust it by subtracting 7. That leaves us with 63 ($3F). We strip off the 3 with AND $0F:

```
    00111111  ($3F, the adjusted ASCII code digit)
AND 00001111  ($0F)
    00001111  (now a true integer F)
```

and then incorporate this F with the $20 we've already got in RESULT from the earlier trip through the loop:

```
    00100000    (RESULT is holding a $20)
ORA 00001111    (we stuff the F into it)
    00101111    (leaving the integer 2F in RESULT)
```

Again, X is raised and tested to see if we're finished with our ASCII hex number (**2340**). This time, we are finished. There's nothing more to roll into RESULT so we set up the HEXFLAG.

This alerts all interested parties in LADS that they do not need to evaluate this argument. The value is already determined and has been placed into RESULT, ready to be printed out or POKEd as the need arises. Then we return to whatever routine called on STARTHEX for its services.

Pseudo-op Preliminaries

The important pseudo-op .BYTE is also handled within the Indisk subprogram. Any pseudo-op beginning with . comes here to PSEUDOJ (**2410**) first. All of these . type pseudo-ops require certain preliminary actions, and the first section of PSEUDOJ accomplishes those things. Then they split up and go to their own specific subroutines. Most of them end up going to the subprogram Pseudo.

PSEUDOJ first tests to see if there is a PC address-type label such as the word OPCODES in:

100 OPCODES .BYTE 161 160 32 96.

The Y Register will still hold a zero if the . character is detected at the very start of a logical line of source code. That would mean that there is no PC-type label and we don't need to bother storing it into the label array for later reference. Likewise, if this isn't pass 1, we can also skip storing such a label in the label array.

But if it is pass 1 and there is one of those labels at the start of the line, we need to save the A and Y Registers (**2450–2470**) and JSR EQUATE to store the PC label (and its address) into LADS' label array. Then we restore the values of A and Y (**2490–2510**) and store the . character in the main input buffer LABEL.

If It's Not B

The character following the . will tell us which pseudo-op we're dealing with, so CHARIN pulls it in and stores it into the buffer (**2550**). If it's not a B, we branch to the springboard PSEUD1 which sends us to the Pseudo subprogram for further tests (**3010**).

Now we know it's a .BYTE type, but is it the ASCII alphabetic type or the ASCII numeric type? It is .BYTE "ABCDE or .BYTE 25 72 1 6?

There is a flag which distinguishes between alphabetic and numeric .BYTEs: the BNUMFLAG. It is first reset (**2600**), and we check both the pass and the SFLAG to decide whether we should print out this line or not. If it's pass 2 and SFLAG is set, we print the line number and the PC address. Then we pull in more of this source code line until we hit a space character. If the character following the space isn't a quote, we know that we're dealing with the numeric type of .BYTE, so we branch down to handle that at BNUMWERK (**2810**).

Otherwise, we take care of the alphabetic type. This type is easy. We can just pull them in and POKE them. There's nothing to figure out or translate. These bytes are held in the source code as ASCII characters and will be POKEd into the object code as ASCII characters. The main use for this pseudo-op is to store messages which will later be printed to the screen or printer.

End-of-Line Alternatives

The active parts of this loop are the CHARIN (**2820**) and the JSR INCSA (**2990**) or JSR POKEIT (**3050**). The decision whether to simply raise the PC with INCSA or actually POKE the object code is based on the test of PASS (**2970**). The rest of the loop (**2830–2960**) is similar to other tests for end-of-line conditions found throughout LADS. We look for a 0 (**2830**), a colon (**2850**), a semicolon (**2880**), and a concluding quote (**2940**). Any of these characters signal the end of our alphabetic message. And each condition exits in a way appropriate to it. Semicolons, for example, require that the comment be stored in BABUF for possible printout. To do this, we JSR PULLREST (**2900**).

PSLOOP stores each character into LABEL, the main input buffer. It also JSRs to the POKEIT routine (in the Printops subprogram) which both stores the character in any object code on disk or memory and raises the PC by 1. Then it jumps back up to the start of the loop to fetch another alphabetic character (**3080**).

Numeric .BYTE

BNUMWERK is more complicated than BY1, the alphabetic .BYTE pseudo-op we just examined. BNUMWERK must not only check for all of those possible end-of-line conditions; it must also *translate* the numbers following .BYTE from ASCII into one-byte integers before they can be POKEd. It's that same problem we've dealt with before: 253 is stored in the source code as three bytes: $32 $35 $33. We need to turn it into a single value: $FD. (One thing simplifies the numeric type .BYTE pseudo-op. The programmer can use only decimal numbers in the source code for this pseudo-op. .BYTE $55 $FF is forbidden, although you could certainly add the option if you wish.)

Like a small version of the Eval subprogram, BNUMWERK has to have a flag which tells it when to close down. We set this BFLAG down (**3100**) and then put the character in the Accumulator into a buffer called NUBUF. In this buffer we'll convert these decimal ASCII numbers into integers. Then we raise X to 1 and enter the main BNUMWERK loop (**3130**).

The BFLAG is tested, and we shut down operations if it is set (**3140**). Otherwise, we pull in the next character and go through that familiar series of tests for end-of-line conditions: 0, colon, or semicolon. If it is a regular character, we stick it into the special BUFM buffer (**3250**) and check to see what pass we're on. On pass 1 we don't do any POKEing or printing out, so we can skip that. But on pass 2, we check to see if we've got a space character, indicating that we've reached the end of a particular number, if not yet the end of an entire line (**3360**). If the number is completely in the buffer, we raise the PC and go back for the next number (**3320**).

On the second pass, however, we may have to POKE object code and also provide printouts. This means that we have to both calculate each number for POKEing as well as store each number in ASCII form for printouts. We pull the character from the BUFM buffer and store it in the printout buffer, LABEL, the main input buffer (**3340**). After that we check again for end-of-number or end-of-line conditions (**3360–3410**) and, not finding one, return for another character (**3440**) after storing the current character in HEXBUF.

An end-of-line condition lands at BSFLAG (**3450**), which alerts BNUMWERK that it should exit the loop after the current number in HEXBUR has been analyzed.

A Huge, and Incorrect, Number

WERK2 (**3480**) performs the analysis of a single number. It points the TEMP variable to NUBUF where the number is stored and JSRs to VALDEC, leaving the value of the number in RESULT. Then the value is POKEd to the disk or RAM object code (and the PC is raised by 1) (**3550**).

So that nothing will be left over to confuse VALDEC during its analysis of the next number, NUBUF is now wiped clean with zeros. VALDEC expects to find 0 at the end of an ASCII number that it's turning into an integer. If that 0 isn't there, VALDEC will keep on looking for it, creating a huge, and incorrect, answer.

Then we return to the main loop and look for another character, the start of another number (**3620**).

Graceful Exits

There are so many options in LADS that graceful exits from routines like BNUMWERK are rather difficult. We cannot just simply RTS somewhere. We've got to take into account several sometimes conflicting conditions.

LADS can get its source code from two places: disk or RAM memory. The source code can be entirely within a single program file or spread across a chain of linked files. LADS can assemble hex or decimal numbers from source code (except within the .BYTE pseudo-op). The assembler can send its object code to four places: disk, screen, RAM memory, or printer. All or any of these targets can be operative at any given time. And output can be turned on or off at will. No wonder there have to be different exits and some testing before we can leave a pseudo-op. We've got to figure out what's expected, where the object code is going. Finally, the fact that logical lines of source code can end in several ways adds one additional complication to the exit.

BBEND is the start of exit testing for BNUMWERK. On pass 1 we have to raise the PC one final number (**3650**). If the line ends with a colon, we cannot go to ENDPRO and look for a new line number, since colons end logical, not physical, lines of source code (**3680**). In either case, we set the COLFLAG up or down, depending on whether or not we've got a colon-type ending to this logical line (**3700**). We then raise the LOCFLAG to tell Eval to print a PC-type address label and PLA PLA, pulling the RTS off the stack in preparation for a JMP back to Eval. If it's pass 1 or if the printer printout flags are down, we don't need to print anything, and we JMP into Eval at STARTLINE to fetch a new line of source code (**3790**).

Alternatively, if it's pass 2 or if the PRINTFLAG is up, we go back into Eval at PRMMFIN where comments following semicolons are printed (**3780**).

FILLDISK (**3810**) takes care of a problem created by using the *= pseudo-op with disk object code files. Recall that if you wrote source code like:

```
10 *= 900
100 START INY
```

110 *= 950; leave room here
120 INX; continue on

LADS would normally store the INY and follow it immediately on a disk file with INX. The PC variable (SA) within LADS would have changed. The INX object code being POKEd to RAM would be stored correctly at address 950. But the INX would go to disk at address 901. The disk is receiving its object code bytes sequentially and doesn't hear about any PC changes within the computer during assembly.

FILLDISK subtracts the old PC value from the new adjusted PC value and sends that number of filler bytes to a disk object file. In the example above, 900 would be subtracted from 950, and 50 bytes would be sent as spacers to the disk. This creates a space between INY and INX, a physical space, which will cause the object file to load into the computer with the correct, expected addresses for each opcode.

A secret is revealed here. There are two full passes, but LADS starts to try for a *third* pass. It is quickly shut down because during this pass the ENDFLAG is up and STARTLINE will detect it. Nevertheless, we cannot store more bytes during this brief condition. Bytes must be stored *only* on pass 2, not on pass 1 or that temporary attempt at a pass 3 (**3840**).

Starting the Countdown

If FILLDISK is called upon to act, however, it acts. The disk object file (file #2) is opened (**3860**), and the old PC is subtracted from the new one (**3880–3940**). The Accumulator is loaded with a 0 and we start the countdown; the result of our subtraction, in the variable WORK, is decremented for each 0 sent to the disk object file (**3960–4000**). If WORK hasn't counted down to zero, we continue with this loop (**4010** and **4030**). Finally, we restore the normal I/O and then return to the caller.

The final subroutine on Indisk is functionally identical to KEYWORD. It turns a token into an ASCII word (turns ? to PRINT), but it sends its results to the BABUF buffer which stores all comments. KEYWORD sends its results to the main buffer LABEL for source code lines. To follow the logic of this subroutine, see the discussion of KEYWORD earlier in this chapter (line 1040 on).

Now we can turn our attention from LADS input to LADS output. The bulk of the next chapter explores the four destinations of assembled code: screen, printer, disk, or memory.

Program 6-1. Indisk

```
10 ; "INDISK" MAIN GET-INPUT-FROM-DISK ROUTINE
20 ;SETUP/EXPECTS DISK TO POINT TO 1ST CHAR IN A NEW LINE (OR BEYOND COLON)
30 ;RESULTS/EITHER FLAGS END OF PROG. OR FILLS LABEL+ WITH LINE OF CODE
40 ;------------------
50 INDISK JSR CLEANLAB; FILL LABEL WITH ZEROS (ROUTINE IN EVAL)
60 LDY #0
70 STY HEXFLAG; PUT HEXFLAG DOWN
80 STY BABFLAG; PUT COMMENTS FLAG DOWN
90 STY BYTFLAG; PUT FLAG SHOWING < OR > DOWN
100 STY PLUSFLAG; PUT ARITHMETIC PSEUDO OP (+) FLAG DOWN
110 LDA COLFLAG; IF THERE WAS A COLON JUST PRIOR TO THIS, REMOVE ANY BLANKS
120 BNE NOBLANKS; (THIS TAKES CARE OF: INY:  LDA 15:    LDX 17 TYPE ERRORS)
130 JSR CHARIN; OTHERWISE, PULL IN THE 1ST CHARACTER (FROM DISK OR RAM)
140 STA LINEN; STORE LOW BYTE OF LINE NUMBER
150 JSR CHARIN
160 STA LINEN+1; STORE HIGH BYTE OF LINE NUMBER
170 NOBLANKS JSR CHARIN; ROUTINE TO ELIMINATE BLANKS FOLLOWING A COLON
175 BNE COOLOOK
176 JSR ENDPRO; THIS HANDLES COLONS PLACED ACCIDENTALLY AT END OF LINE
177 PLA:PLA:JMP STARTLINE
180 COOLOOK CMP #32; (OR FOLLOWING A LINE NUMBER)
190 BEQ NOBLANKS;-------------------------
200 JMP MOI1; SKIP TO CHECK FOR COLON (IT'S EQUIVALENT TO AN END OF LINE 0)
210 STINDISK JSR CHARIN; ENTRY POINT WITHIN LINE (NOT AT START OF LINE)
220 MOINDI BNE MOI1; IF NOT ZERO, LOOK FOR COLON
230 JMP ENDPRO; FOUND A 0 END OF LINE. CHECK FOR END OF PROGRAM (3 ZEROS)
240 MOI1 CMP #58; IS IT A COLON
250 BNE XMO1; IF NOT, CHECK FOR SEMICOLON
260 JMP COLON; FOUND A COLON
```

```
270 XMO1 CMP #59; IS IT A SEMICOLON
280 BNE COMOA; IF NOT CONTINUE ON
290 STY A; FOUND A SEMICOLON (REM)
300 LDA PRINTFLAG; IF PRINTOUT NOT REQUESTED, THEN DON'T STORE THE REMARKS
310 BEQ PULLRX
320 STA BABFLAG; SET UP PRINT COMMENTS FLAG (A MUST BE > Ø AT THIS POINT)
330 LDA A; OTHERWISE, CHECK Y (SAVED ABOVE).  IF ZERO, IS A SEMICOLON AT
340 BEQ PUX; START OF THE LINE (NO LABELS OR MNEMONICS, JUST A BIG COMMENT)
350 JSR PULLREST; OTHERWISE SAVE COMMENTS FOLLOWING THE SEMICOLON
360 JMP MPULL; AND THEN RETURN TO EVAL ---------------------------
370 PUX JSR CHARIN; PUT NON-COMMENT DATA INTO LABEL BUFFER
380 BEQ PUX1; END OF LINE, SO EXIT
390 CMP #127; 7TH BIT NOT SET (SO IT'S NOT A KEYWORD IN BASIC)
400 BCC PUX2
410 JSR KEYWORD; IT IS A KEYWORD, SO EXTEND IT OUT AS AN ASCII WORD
420 PUX2 STA LABEL,Y; PUT THE CHAR. INTO THE MAIN BUFFER
430 INY
440 JMP PUX; RETURN TO LOOP FOR MORE CHARACTERS--------------------
450 PUX1 JSR PRNTLINE; PRINT THE LINE NUMBER
460 JSR PRNTSPACE; PRINT A SPACE
470 JSR PRNTINPUT; PRINT THE CHARACTERS IN THE LABEL BUFFER (MAIN BUFFER)
480 JSR PRNTCR; PRINT A CARRIAGE RETURN
490 LDA #Ø; SET A VARIABLE TO ZERO TO SIGNIFY NOTHING FOR EVAL TO EVALUATE
500 STA A
510 JMP MPULL; GO TO EXIT ROUTINE-----------------------------
520 PULLREST STA BABFLAG; PUT REMARKS INTO BABUF (BUFFER FOR COMMENTS)
530 ;                     THIS ROUTINE REMOVES (AND SAVES) COMMENTS
540 STA A; SET A VARIABLE TO SIGNIFY NOTHING FOR EVAL TO EVALUATE
550 LDY #Ø; SET OFFSET TO BABUF BUFFER FOR FILLING WITH COMMENTS
560 PAX1 JSR CHARIN; GET CHARACTER
570 BNE PAX; IF NOT ZERO, CONTINUE
```

```
580 STA BABUF,Y; OTHERWISE, WE'RE AT THE END OF THE COMMENT
590 LDY A
600 RTS; Y MUST HOLD OFFSET FOR ZERO FILL (ENDPRO)-----------------
610 PAX BPL PAXA; NOT A KEYWORD (7TH BIT NOT SET)
620 JSR KEYWAD; OTHERWISE, EXTEND KEYWORD INTO AN ASCII STRING
630 PAXA STA BABUF,Y; STORE CHAR. IN REMARK BUFFER
640 INY
650 JMP PAX1; RETURN TO LOOP TO GET ANOTHER CHARACTER--------------
660 PULLRX JSR CHARIN; JUST PULL IN REMARK CHARACTERS, IGNORING THEM
670 BEQ MPULL; LOOKING FOR THE END OF LINE ZERO
680 JMP PULLRX;-------------------------
690 MPULL JSR ENDPRO; CHECK FOR END OF PROGRAM AND THEN
700 LDA A; SEE IF Y = 0.  IF SO, THE SEMICOLON WAS AT THE START OF A LINE
710 BNE MPULL1
720 PLA; Y = 0 SO JUMP BACK TO EVAL TO PREPARE TO GET NEXT LINE
730 PLA
740 JMP STARTLINE; SEMI @ START SO RETURN TO EVAL TO GET NEXT LINE---------
750 MPULL1 RTS; SEMICOLON, BUT NOT AT START OF LINE (RETURN TO CALLER)
760 COMOA CMP #177;------------------ CHECK FOR OTHER ODD CHARACTERS
770 BEQ HI; FOUND >
780 CMP #179
790 BEQ LO; FOUND <
800 CMP #170
810 BNE COMO
820 INC PLUSFLAG; FOUND +
830 COMO CMP #172
840 BNE COMO1
850 JMP STAR; FOUND *
860 COMO1 CMP #46
870 BEQ PSEUDOO; FOUND PSEUDO-OP
880 CMP #36
```

```
890 BEQ HEXX; FOUND HEX NUMBER
900 CMP #127; NOT A KEYWORD (7TH BIT NOT UP)
910 BCC ADDLAB
920 JSR KEYWORD; FOUND KEYWORD, SO EXTEND IT INTO AN ASCII STRING
930 ADDLAB STA LABEL,Y; PUT THE CHARACTER INTO THE MAIN BUFFER AND
940 INY; RAISE THE POINTER AND
950 JMP STINDISK; RETURN TO GET ANOTHER CHARACTER (BUT NOT A LINE NUMBER)
960 ;----------------------
970 COLON STA COLFLAG; SIGNIFY COLON BY SETTING COLFLAG
980 RTS;----------------------
990 PSEUDOO JMP PSEUDOJ; SPRINGBOARD TO PSEUDO-OP HANDLING ROUTINES
1000 HEXX STA LABEL,Y; SPRINGBOARD TO HEX NUMBER TRANSLATOR
1010 INY
1020 JMP HEX
1030 ;-------- TRANSLATE A SINGLE-BYTE KEYWORD TOKEN INTO ASCII STRING
1040 KEYWORD SEC; FIND NUMBER OF KEYWORD (IS IT 1ST, 5TH, OR WHAT)
1050 SBC #$7F
1060 STA KEYNUM; STORE NUMBER (POSITION) IN BASIC'S KEYWORD TABLE
1070 LDX #255
1080 SKEY DEC KEYNUM; REDUCE NUMBER BY 1 (WHEN ZERO, WE'VE FOUND IT IN TABLE)
1090 BEQ FKEY; AND WE EXIT THIS SEARCH ROUTINE AND STORE THE ASCII WORD
1100 KSX INX; BRING X UP TO ZERO AT START OF LOOP
1110 LDA KEYWDS,X; LOOK AT CHAR. IN BASIC'S TABLE.
1120 BPL KSX;DID NOT FIND A SHIFTED BYTE
1130 BMI SKEY; DID FIND START-OF-KEYWORD SHIFTED CHARACTER ------------
1140 FKEY INX; STORE THE KEYWORD INTO LADS' MAIN BUFFER (LABEL)
1150 LDA KEYWDS,X
1160 BMI KSET; A SHIFTED CHAR. INDICATES END OF KEYWORD, START OF NEXT KEYWORD
1170 STA LABEL,Y; PUT CHAR. INTO LADS' BUFFER
1180 INY
1190 JMP FKEY; LOOP AGAIN FOR NEXT CHAR.---------------------
```

```
1200 KSET AND #$7F
1210 RTS; CLEAR OUT BIT 7 AND RETURN TO CALLING ROUTINE
1220 ;------------------- HANDLE > AND < PSEUDO-OPS
1230 HI LDA #2;        THE BYTFLAG HAS 3 POSSIBLE STATES:
1240 STA BYTFLAG;            0 = LINE DOESN'T CONTAIN A > OR < PSEUDO
1250 JMP STINDISK;           1 = < (LOW BYTE) TYPE
1260 LO LDA #1;              2 = > (HIGH BYTE) TYPE
1270 STA BYTFLAG;          (ACTION IS TAKEN ON THIS PSEUDO-OP WITHIN THE
1280 JMP STINDISK;           EQUATE SUBPROGRAM).  O WE FETCH THE NEXT CHAR.
1290 ;------------------- HANDLE THE *= PSEUDO-OP (CHANGE THE PC)
1300 STAR JSR STINDISK
1310 ;LDA PASS; ON PASS 1, DON'T PRINT OUT DATA TO SCREEN
1320 ;BEQ STARN
1325 LDA #$18:JSR PRINT
1330 LDA #42; PRINT *
1340 JSR PRINT
1350 JSR PRNTINPUT; PRINT STRING IN LABEL BUFFER
1360 JSR PRNTCR; PRINT CARRIAGE RETURN
1370 STARN LDA HEXFLAG; IF HEX, THE ARGUMENT HAS ALREADY BEEN FIGURED
1380 BNE STARR; SO JUMP OVER THIS NEXT PART
1390 LDY #0
1400 STAF LDA LABEL,Y
1410 CMP #32
1420 BEQ STAF1
1430 INY
1440 JMP STAF; FIND NUMBER (BY LOOKING FOR THE BLANK: *= 15)
1450 STAF1 INY
1460 STY TEMP; POINT TO ASCII NUMBER
1470 LDA #<LABEL
1480 CLC
```

```
1490 ADC TEMP
1500 STA TEMP
1510 LDA #>LABEL
1520 ADC #0
1530 STA TEMP+1
1540 JSR VALDEC; TRANSLATE ASCII NUMBER INTO INTEGER (IN RESULT)
1550 STARR LDA PASS; ON PASS 1, LEAVE DISK OBJECT FILE ALONE.
1560 BEQ STARRX
1570 LDA DISKFLAG; ON PASS 2, WE'VE GOT TO STUFF THE DISK OBJECT FILE
1580 BEQ STARRX; IF THE DISKFLAG IS UP (WE ARE CREATING AN OBJECT CODE FILE)
1590 JSR FILLDISK; FILLDISK DOES THIS FOR US.
1600 STARRX LDA RESULT; PUT THE ARGUMENT OF *= INTO THE PC (SA)
1610 STA SA
1620 LDA RESULT+1
1630 STA SA+1
1640 PLA; PULL OFF THE RTS AND
1650 PLA
1660 JMP STARTLINE; RETURN TO EVAL FOR THE NEXT LINE OF CODE
1670 ;------------------ IS THIS THE END OF THE ENTIRE SOURCE CODE
1680 ENDPRO STA LABEL,Y; PUT THE ZERO (THAT SENT US HERE) INTO THE MAIN BUFFER
1690 INY
1700 CPY #80
1710 BNE ENDPRO; FILL REST OF BUFFER WITH 00S
1720 STA LABEL,Y
1730 JSR CHARIN; PULL IN THE NEXT 2 BYTES.  IF THEY ARE BOTH ZEROS, THEN
1740 JSR CHARIN; WE HAVE, IN FACT, FOUND THE END OF OUR SOURCE CODE FILE
1750 BEQ INEND; AND WE BEQ TO INEND
1760 LDA #0; OTHERWISE WE PUT THE COLFLAG (COLON) DOWN, BECAUSE THIS IS
1770 STA COLFLAG; AN END OF LINE CONDITION, NOT A COLON
1780 RTS; AND RETURN TO CALLER
1790 INEND LDA #1;-------- SET END OF SOURCE CODE FILE FLAG TO UP CONDITION
```

```
1800 STA ENDFLAG
1810 RTS; AND RETURN TO CALLER
1820 ;----------------------- CHANGE A HEX NUMBER TO A 2-BYTE INTEGER
1830 ; PULL IN NEXT FEW BYTES, TURNING THEM INTO AN INTEGER IN RESULT
1840 HEX LDX #0; PUTS INTEGER EQUIVALENT OF INCOMING HEX INTO RESULT
1850 H1 JSR CHARIN
1860 BEQ DECI; END OF LINE (SO STOP LOOKING)
1870 CMP #58
1880 BEQ DECI; COLON (SO STOP LOOKING)
1890 CMP #32
1900 BEQ H1; BLANK CHARACTER SO KEEP LOOKING FOR END OF LINE
1910 CMP #59
1920 BEQ DECI; SEMICOLON (SO STOP LOOKING)
1930 CMP #44
1940 BEQ DECIT; COMMA (SO STOP LOOKING, BUT GO TO A DIFFERENT PLACE)
1950 CMP #41; (THIS "DIFFERENT PLACE" HANDLES A NOT-END-OF-LINE CONDITION).
1960 BEQ DECIT; CLOSE PARENTHESIS ) (SO STOP LOOKING)
1970 STA HEXBUF,X; OTHERWISE, PUT THE ASCII-STYLE-HEX CHAR. IN BUFFER AND
1980 INX; RAISE THE INDEX AND
1990 STA LABEL,Y; ALSO STORE IT INTO MAIN BUFFER FOR PRINTOUT AND
2000 INY; RAISE THIS INDEX TOO
2010 JMP H1; THEN KEEP ON PUTTING HEX NUMBER INTO HEXBUFFER----------
2020 DECIT STX HEXLEN; SAVE LENGTH OF ASCII-HEX NUMBER
2030 STA LABEL,Y; FINISH STORING CHARS. INTO MAIN BUFFER (, OR ) IN THIS CASE)
2040 INY
2050 JSR STARTHEX; TRANSLATE ASCII-HEX NUMBER INTO INTEGER IN RESULT VARIABLE
2060 JMP STINDISK; RETURN TO PULL IN REST OF THE LINE;----------
2070 DECI STA A; SAVE THE END OF LINE, COLON, OR SEMICOLON CHAR. FOR LATER
2080 LDA #0
2090 STX HEXLEN; SAVE LENGTH OF ASCII-HEX NUMBER
2100 STA LABEL,Y; FINISH STORING CHARS. INTO MAIN BUFFER (0 IN THIS CASE)
```

```
2110 JSR STARTHEX; TRANSLATE ASCII-HEX NUMBER INTO INTEGER IN RESULT VARIABLE
2120 LDA A; RETRIEVE Ø OR COLON OR SEMICOLON AND GO BACK UP TO MOINDI WHICH
2130 JMP MOINDI;----------------- BEHAVES ACCORDING TO WHICH SYMBOL A HOLDS.
2140 STARTHEX LDA #Ø;----------------- HEX-ASCII TO INTEGER TRANSLATOR------
2150 STA RESULT; SET RESULT TO ZERO
2160 STA RESULT+1
2170 TAX; SET X TO ZERO
2180 HXLOOP ASL RESULT; SHIFT AND ROLL (MOVES 2-BYTE BITS TO THE LEFT)-----
2190 ROL RESULT+1; DOING THIS 8 TIMES HAS THE EFFECT OF BRINGING IN
2200 ASL RESULT; THE ASCII NUMBER, 1 BYTE AT A TIME, AND TRANSFORMING IT
2210 ROL RESULT+1; INTO A 2-BYTE INTEGER WITHIN THIS 2-BYTE VARIABLE WE'RE
2220 ASL RESULT; CALLING "RESULT."
2230 ROL RESULT+1
2240 ASL RESULT
2250 ROL RESULT+1
2260 LDA HEXBUF,X; GET A BYTE FROM THE ASCII-HEX NUMBER
2270 CMP #65; IF IT'S LOWER THAN 65, IT'S NOT AN ALPHABETIC (A-F) HEX NUMBER
2280 BCC HXMORE; SO DON'T SUBTRACT 7 FROM IT
2290 SBC #7; BUT IF IT'S > 65, THEN -7.   = 65.  65-7 = 58.
2300 HXMORE AND #15; WHEN YOU 58 AND 15, YOU GET 1Ø (THE VALUE OF A)
2310 ORA RESULT; #15 (ØØØØ1111) AND #58 (ØØ111Ø1Ø)  = ØØØØ1Ø1Ø (TEN)
2320 STA RESULT; PUT THE BYTE INTO RESULT
2330 INX; RAISE THE INDEX
2340 CPX HEXLEN; ARE WE AT THE END OF OUR ASCII-HEX NUMBER
2350 BNE HXLOOP; IF NOT, CONTINUE
2360 INC HEXFLAG; IF SO, RAISE HEXFLAG (TO SHOW RESULT HAS THE ANSWER)
2370 LDA #1; AND RETURN TO CALLER
2380 RTS
2390 ;----------------------
2400 ; HANDLE PSEUDOS.  (.BYTE TYPES)
2410 PSEUDOJ CPY #Ø; IF Y = Ø THEN IT'S NOT A PC LABEL LIKE (LABEL .BYTE Ø Ø)
```

```
2420 BEQ PSE2
2430 LDX PASS; OTHERWISE, ON 1ST PASS, STORE LABEL NAME AND PC ADDR. IN ARRAY
2440 BNE PSE2
2450 PHA; SAVE A AND Y REGISTERS
2460 TYA
2470 PHA
2480 JSR EQUATE; NAME AND PC ADDR. STORED IN ARRAY
2490 PLA; PULL OUT A AND Y REGISTERS (RESTORE THEM)
2500 TAY
2510 PLA
2520 PSE2 STA LABEL,Y; STORE . CHAR.
2530 INY
2540 JSR CHARIN; GET CHAR. FOLLOWING THE PERIOD (.)
2550 STA LABEL,Y
2560 INY
2570 CMP #66; IS IT "B" FOR .BYTE
2580 BNE PSEUD1; WASN'T .BYTE
2590 LDA #0; RESET FLAG WHICH WILL DISTINGUISH BETWEEN .BYTE 0 AND .BYTE "A
2600 STA BNUMFLAG; " TYPE, OR 00 08 15 172 TYPE (THE TWO .BYTE TYPES)
2610 LDA PASS; PRINT NOTHING TO SCREEN ON PASS 1
2620 BEQ CLB
2630 STY Y; SAVE Y REGISTER (OUR INDEX)
2640 ;                     NOW WE REPLICATE THE ACTIONS OF INLINE (IN EVAL)
2650 LDA SFLAG; SHOULD WE PRINT TO SCREEN
2660 BEQ CLB; NO
2670 JSR PRNTLINE; YES, PRINT LINE NUMBER
2680 JSR PRNTSPACE; PRINT SPACE
2690 JSR PRNTSA; PRINT PC ADDRESS
2700 JSR PRNTSPACE; PRINT SPACE
2710 LDY Y; RECOVER Y INDEX
2720 CLB JSR CHARIN; PULL IN CHARACTER FROM DISK/RAM
```

```
2730 STA LABEL,Y; STORE IN MAIN BUFFER
2740 INY
2750 CMP #32; IS IT A SPACE
2760 BNE CLB; IF NOT, CONTINUE PULLING IN MORE CHARACTERS------------
2770 JSR CHARIN; (WE'RE LOOKING FOR THE 1ST SPACE AFTER .BYTE)
2780 STA LABEL,Y; STORE FOR PRINTING
2790 INY
2800 CMP #34; IS THE CHARACTER A QUOTE ("). IF SO, IT'S A .BYTE "ABCD TYPE
2810 BNE BNUMWERK; OTHERWISE IT'S NOT THE " TYPE
2820 BY1 JSR CHARIN;----------  HANDLE ASCII STRING .BYTE TYPES
2830 BNE BY2
2840 JMP BENDPRO; FOUND A 0 END OF LINE (OR PROGRAM)
2850 BY2 CMP #58; FOUND A COLON "END OF LINE"
2860 BNE BY2X
2870 JMP BEN1; FOUND A COLON
2880 BY2X CMP #59; FOUND A SEMICOLON "END OF LINE"
2890 BNE BY3
2900 JSR PULLREST; STORE COMMENTS IN COMMENT BUFFER (BABUF)
2910 LDX PRINTFLAG; IF NO PRINTOUT REQUESTED, THEN
2920 STX BABFLAG; DON'T PRINT COMMENTS
2930 JMP BENDPRO; A SEMICOLON SO END THIS ROUTINE IN THAT WAY.
2940 BY3 CMP #34; HAVE WE FOUND A CONCLUDING QUOTE (")
2950 BNE BY3X
2960 JMP BY1; FOUND A " SO IGNORE IT
2970 BY3X LDX PASS; ON PASS 1, JUST RAISE PC COUNTER (INCSA); DON'T POKE IT.
2980 BNE PSLOOP
2990 JSR INCSA
3000 JMP BY1;-----------------
3010 PSEUD1 JMP PSEUDO; SOME OTHER PSEUDO TYPE, NOT .BYTE (A SPRINGBOARD)
3020 PSLOOP STA LABEL,Y; STORE A CHARACTER IN MAIN BUFFER;------------
3030 TAX
```

```
3040 STY Y; SAVE Y INDEX
3050 JSR POKEIT; PASS 2, SO POKE IT INTO MEMORY (THE ASCII CHARACTER)
3060 LDY Y; RESTORE Y
3070 INY; RAISE INDEX AND
3080 JMP BY1; GET NEXT CHARACTER
3090 BNUMWERK LDX #0;-------- HANDLE .BYTE 1 2 3 (NUMERIC TYPE)
3100 STX BFLAG; PUT DOWN BFLAG (END OF LINE SIGNAL)
3110 STA NUBUF,X; WE'RE BORROWING THE NUBUF FOR THIS ROUTINE.
3120 INX
3130 WERK1 LDA BFLAG; IF BFLAG IS UP, WE'RE DONE.
3140 BNE BBEND; SO GO TO END ROUTINE
3150 WK0 JSR CHARIN; OTHERWISE, GET A CHARACTER FROM DISK/RAM
3160 BEQ BSFLAG; IF ZERO (END OF LINE) SET BFLAG UP.
3170 CMP #58; LIKEWISE IF COLON
3180 BEQ BSFLAG
3190 CMP #59; SEMICOLON REQUIRES THAT WE FIRST FILL THE COMMENT BUFFER
3200 BNE WK1;         BEFORE SETTING THE BFLAG (IN THE BSFLAG ROUTINE)
3210 JSR PULLREST; HERE'S WHERE THE COMMENT BUFFER IS FILLED
3220 LDX PRINTFLAG; IF NO PRINTOUT REQUESTED, THEN
3230 STX BABFLAG; DON'T PRINT COMMENTS
3240 JMP BSFLAG; FOUND SEMICOLON
3250 WK1 STA BUFM; PUT CHAR. INTO "BUFM" BUFFER
3260 LDA PASS; ON PASS 1, RAISE THE PC ONLY (INCSA), NO POKES
3270 BNE WERK5
3280 LDA BUFM
3290 CMP #32; IS IT A SPACE
3300 BNE WERK1; IF NOT, RETURN FOR MORE OF THE NUMBER (0 VS 555)
3310 JSR INCSA; RAISE PC COUNTER BY 1
3320 JMP WERK1; GET NEXT NUMBER
3330 WERK5 LDA BUFM; PUT CHAR. INTO PRINTOUT MAIN BUFFER
3340 STA LABEL,Y
```

```
3350 INY
3360 CMP #32; IS IT A SPACE
3370 BEQ WERK2
3380 CMP #0; IS IT END OF LINE
3390 BEQ WERK2
3400 CMP #58; IS IT COLON
3410 BEQ WERK2
3420 STA NUBUF,X; OTHERWISE, STORE IT
3430 INX
3440 JMP WERK1; AND RETURN FOR MORE OF THE NUMBER----------------
3450 BSFLAG INC BFLAG; RAISE UP THE END OF LINE FLAG
3460 STA BUFM+1; SAVE COLON, SEMICOLON, OR WHATEVER FOR LATER USE
3470 JMP WK1; RETURN FOR MORE (BUT THIS TIME IT WILL END LINE);---------
3480 WERK2 LDA #<NUBUF; POINT TO THE ASCII NUMBER STORED IN BABUF
3490 STA TEMP
3500 LDA #>NUBUF
3510 STA TEMP+1
3520 STY Y
3530 JSR VALDEC; TURN THE ASCII INTO AN INTEGER IN RESULT
3540 LDX RESULT
3550 JSR POKEIT; POKE THE RESULT INTO MEMORY (OR DISK OBJECT FILE)
3560 LDY Y; ERASE THE NUMBER IN HEXBUF
3570 LDA #0
3580 LDX #5
3590 CLEX STA NUBUF,X
3600 DEX
3610 BNE CLEX
3620 JMP WERK1; AND THEN RETURN TO FETCH THE NEXT NUMBER;--------------
3630 BBEND LDA PASS; END .BYTE LINE.  ON PASS 1, RAISE PC (POKEIT RAISES IT
3640 BNE BBEND1;                      ON PASS 2).
3650 JSR INCSA
```

```
3660 BBEND1 LDA BUFM+1; IF END OF LINE SIGNAL WAS A COLON, THEN
3670 CMP #58
3680 BEQ BEN1; DON'T LOOK FOR LINE NUMBER OR END OF SOURCE CODE FILE (ENDPRO)
3690 BENDPRO JSR ENDPRO
3700 BEN1 STA COLFLAG; SET IT (COLON) OR NOT (ENDPRO RETURNS WITH Ø IN A)
3710 INC LOCFLAG; RAISE PRINT-A-PC-LABEL FLAG
3720 PLA; PULL RTS FROM STACK
3730 PLA
3740 LDA PASS; ON PASS 1, DON'T PRINT ANY COMMENTS
3750 BEQ NOPR
3760 LDA SFLAG; IF SCREENFLAG IS DOWN, DON'T PRINT ANY COMMENTS
3770 BEQ NOPR
3780 JMP PRMMFIN; BACK TO EVAL (WHERE COMMENTS ARE PRINTED)
3790 NOPR JMP STARTLINE; BACK TO EVAL (BYPASSING PRINTOUT)
3800 ;----------------------  FOR CHANGE OF PC
3810 FILLDISK LDA PASS; A CHANGE OF PC REQUIRES FILLING A DISK OBJECT FILE
3820 CMP #2;            WITH THE REQUISITE NUMBER OF BYTES TO MAKE UP FOR
3830 BNE FILLX;         THE ADVANCING OF THE PROGRAM COUNTER (PC)
3840 RTS; NOT AT START OF 3RD PASS (3RD PASS IS JUST BEFORE SHUT DOWN)
3850 FILLX JSR CLRCHN
3860 LDX #2
3870 JSR CHKOUT; PUT SPACERS IN DISKFILE FOR *=
3880 SEC; FIND OUT HOW MANY SPACERS TO SEND TO DISK BY SUBTRACTING:RESULT-SA
3890 LDA RESULT
3900 SBC SA
3910 STA WORK; ANSWER HELD IN "WORK" VARIABLE
3920 LDA RESULT+1
3930 SBC SA+1
3940 STA WORK+1
3950 PUTSPCR LDA #Ø
3960 JSR PRINT; PRINT SPACER TO DISK
```

```
3970 LDA WORK; LOWER WORK BY 1
3980 BNE DECWORKX
3990 DEC WORK+1
4000 DECWORKX DEC WORK
4010 BNE PUTSPCR
4020 LDA WORK+1
4030 BNE PUTSPCR; PUT MORE SPACERS IN UNTIL "WORK" IS DECREMENTED TO ZERO.
4040 RESFILL JSR CLRCHN
4050 LDX #1; RESTORE NORMAL I/O
4060 JSR CHKIN
4070 RTS
4080 ;----------------------
4090 KEYWAD SEC; SEE KEYWORD ABOVE (SAME KEWORD TO ASCII STRING ROUTINE)
4100 SBC #$7F; THIS IS A VERSION OF KEYWORD, BUT FOR COMMENTS(PUTS IT IN BABUF
4110 STA KEYNUM;                   INSTEAD OF LABEL BUFFER).
4120 LDX #255
4130 SKEX DEC KEYNUM
4140 BEQ FKEX
4150 KSXX INX
4160 LDA KEYWDS,X
4170 BPL KSXX
4180 BMI SKEX
4190 FKEX INX
4200 LDA KEYWDS,X
4210 BMI KSEX
4220 STA BABUF,Y
4230 INY
4240 JMP FKEX
4250 KSEX AND #$7F
4260 RTS
4270 ;-----------------
```

```
4280 .FILE MATH
```

Program 6-2. Indisk, Apple Modifications

To create the Apple version of Indisk, change the following lines in Program 6-1:

```
740 COMOA CMP #$3E;----------------- CHECK FOR OTHER ODD CHARACTERS
760 CMP #$3C
780 CMP #$2B
810 COMO CMP #$2A
830 CPY #255
```

Program 6-3. Indisk, Atari Modifications

To create the Atari version of Indisk, omit lines 1040–1210 and lines 4090–4260 of Program 6-1 and add or change the following lines:

```
10 ;ATARI MODIFICATIONS--INDISK
115 JSR LINENUMBER
390 ;
400 ;
410 ;
610 PAX NOP
620 ;
760 COMOA CMP #62
780 CMP #60
800 CMP #43
830 COMO CMP #42
900 ;
```

```
910 ;
920 ;
1325 ;
1730 LDA $0353
1740 CMP #$03
1751 CMP #136
1752 BEQ INEND
4280 .FILE D:MATH.SRC
```

Chapter 7

Math and Printops:

Range Checking and Formatted Output

Math and Printops: Range Checking and Formatted Output

Math, a short subprogram, has a rather limited job. It is designed to turn the ASCII number following the + pseudo-op into a two-byte integer and to save it in the variable ADDNUM. Later, when the final RESULT is calculated by the Valdec subprogram, anything in ADDNUM will be added to RESULT. Math responds to a source code line like:

```
100 SCREEN = $0400
120 LDA SCREEN+256; this would assemble as $0500
```

As with the .BYTE pseudo-op, the + pseudo-op allows only decimal numbers as an argument following the +.

The first loop in the Math subprogram simply looks along the LABEL buffer to locate the +. Thus, it doesn't matter if the + is right next to its label. You could write SCREEN   +256 as well as SCREEN+256. However, finding the +, the subroutine expects to find no spaces between the + and the number to be added. +256 is correct. + 256 would be incorrect. This allows us to test for a variety of end-of-number conditions. That means that you can use the + pseudo-op within such addressing modes as LDA (SCREEN+256),Y or LDA 1500+25,Y.

Each character following the + is stored in HEXBUF for later translation by Valdec. Each is also tested to see if it is a nonnumber—if it is outside the range from 47 to 58, the ASCII code for the digits 0–9. Anything outside that range ends our storage of the number to be added, and we go down to put the number into ADDNUM.

Range checking is simple enough. Just remember to test against a number which is one lower than the low end and one higher than the high end of the range. For example, to see if a number is lower than $30, you must test against $2F. That's because BCC tests for *lower than*. $30 wouldn't be lower than $30. The same thing works on the high end. To test for numbers higher than $39, you CMP #$3A.

After the number is set up in HEXBUF, we point TEMP to it, JSR to Valdec, and move the result from RESULT into the variable ADDNUM. It will wait there until, on pass 2, the Array subprogram makes the addition adjustment in line 1160.

Printops: The Output Routine

One important function performed by the Printops subprogram is raising the PC (Program Counter). A subroutine called INCSA (**650**) increases the PC by one for each object code byte, whether this byte is an opcode or the argument of an opcode. Printops' other main job is to send each byte of object code to one of four places: RAM memory, disk, screen, or printer.

Because each object code byte can go to any one, or all, of these four different destinations, there are a series of tests and parallel routines within Printops. For one thing, Printops has little to do on pass 1—it does raise the PC, but nothing is POKEd anywhere or printed to screen or printer until the second pass.

Also, Printops has three entry points, depending on whether the Eval subprogram has assembled a one-, two-, or three-byte logical line. An INY would only JSR from Eval to FORMAT, right at the start of Printops. FORMAT loads the OP (opcode) and stores it and prints it as required. It's a single-byte event. LDA 15 first JSRs to FORMAT to output the opcode, the numeric equivalent of LDA, then enters at PRINT2. LDA 1500 would JSR FORMAT to send the opcode, then enter at PRINT3. These entry decisions are made by Eval after it has determined whether it's dealing with a one-, two-, or three-byte addressing mode.

FORMAT (**20**) simply raises the PC by one. It does this with a JSR to INCSA (**40**) on pass 1. On pass 2, however, it also checks to see if screen printout was requested (**60**). If so, it restores normal I/O and prints the number (**120**). As we will see, PRINTNUM also prints to the printer, if that was requested. Then the opcode is POKEd to disk or RAM, if that was requested. The POKEIT subroutine performs POKEs to RAM. POKEIT also leads right into INCSA to raise the PC automatically following each POKE. Finally we RTS back to Eval (**160**). So much for a single-byte addressing mode.

Two-Byte Addressing Modes

PRINT2 (**180**) handles LDA 15 and other two-byte addressing modes. Like FORMAT, pass 1 only results in a JSR INCSA (to raise the PC). Pass 2 follows the same pattern as FORMAT, explained above. The major difference is that the number fetched before the JSR to PRINTNUM comes from the low byte of the RESULT variable (**240**) rather than OP. This is a single-byte argument addressing mode.

PRINT3 (**290**) parallels the two previous routines, except that it handles a two-byte argument. On pass 1 it JSRs to INCSA twice to raise the PC by two.

On pass 2, it prints (**370**) and POKEs (**390**) the low byte of RESULT if requested and then prints (**460**) and POKEs (**480**) the high byte of the argument, RESULT+1. A formatting problem is handled in line 420. HXFLAG shows whether or not output to screen and printer is supposed to be in hex. If this flag is set, we don't need to print a space between the low and high bytes of the argument. The hex printing routine will do that for us. If printout is in decimal, though, we need to print a space (**440**).

Creating an Object Program

POKEIT (**490**) stores the byte in the X Register at the current PC address if the POKEFLAG is up. This flag indicates that the programmer used the .O pseudo-op, requesting that object code be stored in RAM memory during assembly. For both PRINTNUM and POKEIT, the X Register is holding the opcode or argument. X is saved in the variable WORK+1; some of the disk management routines below will change the value of X, so we must preserve it for later use.

Then the DISKFLAG is checked (**550**). It indicates that the programmer used the .D pseudo-op, asking that an object code program file be created on disk during assembly. If not, we just go down to raise the PC at INCSA (**560**).

But if an object program *is* being created on disk, LADS opens communication to file #2 (the write-to-disk file) and recovers the byte from WORK+1 (**600**). The PRINT in 610 will not go to screen or printer. Rather, the current channel is open to the disk object file and PRINT therefore sends the byte in the Accumulator to the disk. Then normal I/O is restored, and file #1 is accessed again. File #1 is the normal input source for LADS, the read-from-disk channel. Finally, we fall through to INCSA (**650**).

Although it is one of the simplest events in LADS, INCSA is also one of the most important. On both passes, INCSA raises the PC by 1 for each opcode byte and for each argument byte. Much depends on the fact that INCSA keeps the Program Counter accurate during assembly. A single ignored byte would throw off all address-type labels which followed. (The HERE in 100 HERE LDA 15 is an address-type label.) In consequence, the entire assembled object program would be useless. INCSA just adds 1 to SA (the variable which holds the LADS internal Program Counter). Notice lines 690–710. They add 0 to the high byte of SA. What's the point of that?

The 256th Increment

For every 255 increments, INCSA will have nothing to add to the high byte of SA. But on the 256th increment, it must add 1 to the high byte. How does adding 0 to the high byte add 1 to it? The carry flag. ADC means ADd with Carry. If the carry flag is set, the high byte is incremented. If the low byte is holding 255 when we add 1 to it (**670**), that will set the carry flag.

The rest of the routines in this Printops subprogram handle the printout of a variety of things: messages, spaces, numbers, the PC address, a carriage return, a source code line number, a source code line, or an error message. And each of these print-to-screen routines has a sister routine. There is a parallel series of routines which print the same thing to the printer.

PRNTMESS (**740**) will print any ASCII message. There are two special requisite preconditions: The message must be pointed to by the variable TEMP, and the message must end with a 0. PRNTMESS is a simple loop, but it can print any message you want. First the Y Register is set to 0 to act as an index to the message within LADS' source code. Then the loop begins (**750**) by loading in a character from the message (**750**). If the character is 0, we exit the loop. Otherwise, the character is printed to the screen. Then we JSR to the sister routine, PTP, which will send the same character to the printer, if requested (**780**). The Y Register is raised, and we go back for the next character (**800**).

PRINTSPACE (**820**) simply prints a space character to the screen and then checks with its sister routine, PTP, to see if the space should also be printed on the printer.

Before printing a number, we first put it into the X variable for safekeeping. Then LADS has to make four tests: Is it printout to screen or to printer, and is it in decimal or in hex numbers? PRNTNUM (**860**) takes advantage of a routine in BASIC ROM if

LADS' printout is in decimal (requested with the .NH, no hex, pseudo-op). When you ask BASIC to list a program, it turns integer bytes into printable ASCII numbers to provide line numbers on the screen. On Commodore computers, the high byte of the integer is put into the Accumulator, the low byte into the X Register, and you JSR to within BASIC ROM where this routine resides (**950**). In LADS, the address of this ROM routine is called OUTNUM. It's defined for each different computer model in the Defs subprogram.

Hex Default

LADS' default, and probably the most common way to print out numbers during an assembly, is hex. LADS itself handles hex printing. If the HXFLAG is up (**870**), we JSR to HEXPRINT, a subroutine at the end of the Printops subprogram. We'll get to it in a minute. It's the opposite of the HEX subroutine within the Indisk subprogram which changes hex numbers in ASCII format into integers. The HEXPRINT routine will take an integer and turn it into hex ASCII characters for printout.

After the number has been printed to the screen, we JSR to the sister routine PTPNU (**910**) to also print it to the printer if necessary. Then the number is restored to the X Register from the X variable (**920**) before returning to the caller.

PRNTSA (**990**) is similar to PRNTNUM. The main difference is that PRNTNUM always prints the single byte sent to it in the X Register. By contrast, PRNTSA prints the two bytes in SA, the Program Counter variable. The same four possibilities are tested: printer, screen, hex, or decimal. PRNTSA's sister routine, PTPSA, is called upon from both the hex (**1050**) and the decimal (**1100**) versions of this routine.

PRNTCR (**1120**) prints a carriage return; the 13 is the ASCII code for carriage return on both the screen and a printer. PRNTLINE (**1160**) prints out a line number from the source code. As each physical line is drawn into view by LADS, its line number is stored in the LINEN variable. This routine also uses that OUTNUM routine from BASIC ROM which prints BASIC's line numbers during a LIST. Line numbers, in BASIC or LADS, are always decimal. PTPLI (**1190**) is the sister routine for printer printouts.

PRNTINPUT (**1210**) prints the contents of the main buffer. Those contents will be the most recent logical line of source code *as it appeared in the source code.* It uses the PRNTMESS routine which sends to the screen any ASCII message which is pointed to by the TEMP variable. The line must end in 0. PRNTMESS (**740**) handles the printer with the PTP, single-character, test. There is no need for a sister routine within PRNTINPUT.

Error Alert

ERRING (**1280**) performs the preliminaries to an error message printout. Such messages as SYNTAX ERROR or NAKED LABEL are triggered at various places within LADS. But most of them JSR to ERRING before printing out their particular messages. ERRING rings the bell first. The number 7 is the ASCII code which rings any bells attached to computers or printers. (This works on Apple and PET/CBM computers; the 7 is changed to 253 in the Atari version to produce the same result. The VIC and Commodore 64 have no "bell," so the character 7 will have no effect on those computers.) The purpose of the bell is to alert the programmer that an error has been detected. True, the error message will appear onscreen, but during an assembly of a large program, the programmer might well miss silent error messages sliding up the screen.

On Commodore computers, the character 18 reverses the field of all subsequent characters on a line. This, too, highlights errors. Next (**1320**), the logical line of source code where the error appears is printed, followed by a carriage return.

It would be simple to make error reports more dramatic. You could stop assembly at that point with a key-testing loop that required the programmer to hit any key to continue. You could JSR FIN and exit to BASIC mode, aborting all further assembly. You could JSR PRNTLINE to emphasize the line number in the source code where the error happened. You could ring the bell ten times. As with all other aspects of LADS, you can make it do what's efficient for you, what's responsive to your own style of programming. Add some special effects here if you wish. Then reassemble your customized version of LADS.

Sister Print Routines

The next few routines are the printer routines: Each is a parallel, sister routine to one of the screen routines discussed above. Each tests the PRINTFLAG and returns if the flag is down, indicating that the user did not request a printout on paper. If the PRINTFLAG is up, output is redirected to the printer (**1450–1470**) by opening a file channel to the printer. On Commodore computers, the printer is device #4. Then OUTNUM or PRINT or HEXPRINT sends the characters or numbers to the printer (**1490, 1680, 1720, 1900, 1960, 2130**). After that, normal I/O is restored (**1500**) and a channel is reopened to file #1, the input-source-code-from-disk mode.

To follow the logic of PTP (**1380**), PTPNU (**1560**), PTPSA (**1780**), or PTPLI (**2020**), just look at the parallel routines which JSR to them. The purpose, the tests, and the logic are the same. The only difference is that the sister routines described above route their characters to the screen. These routines send characters to a printer.

Printing Hex Numbers

The subprogram Printops concludes with HEXPRINT, an interesting routine which converts a one-byte integer into an ASCII hex string that can be printed to screen or printer.

HEXPRINT operates on a single byte at a time. The byte is first stored temporarily on the stack with PHA (**2200**). Let's use $4A as an example. The four high bits are stripped off with AND #$0F, leaving $0A. That's one of the characters we need to print. Then we can use a short, simple lookup table to extract the character by its position in the table. In the Tables subprogram is a minitable called HEXA (**270**). It looks like this:

```
270 HEXA .BYTE "0123456789ABCDEF
```

Since the number $0A (10 decimal) is also the tenth character in this table, we can just move the ANDed $0A over to the Y Register (**2220**) and load HEXA,Y to fetch the ASCII character for $0A, which would be 65 (the letter A). We can stick this character into the X Register; X isn't being used elsewhere in this routine, so it can save the character for us while we look at the high bits.

this time we move the four high bits right over on top of the four low bits. This takes four logical shifts right (**2270–2300**). After LSRing $4A we get $04. Again, we TAY and load the character 4 from the table (it's 52 decimal). We print this. In $4A, the 4 comes first. Then we recover the A character from the X Register and print it right after the 4 (**2350**).

Program 7-1. Math

```
10 ; "MATH"  THIS ROUTINE HANDLES +   IT COMES FROM EVAL AFTER INDISK
20 ; IT LEAVES THE INTENDED ADDITION IN THE VARIABLE "ADDNUM"
30 ; (ADDNUM IS ADDED TO "RESULT" IN THE VALDEC SUBPROGRAM)
40 MATH LDY #Ø; SET INDEXES TO ZERO
50 LDX #Ø
60 MATH1 LDA LABEL,Y; LOOK FOR LOCATION OF "+" SYMBOL--------
70 CMP #43
80 BEQ MATH2
90 INY
100 JMP MATH1;------------- NOW POINT TO 1ST NUMBER FOLLOWING +
110 MATH2 INY
120 LDA LABEL,Y
130 JSR RANGECK; CHECK TO SEE IF THIS IS BETWEEN 48 - 58 (ASCII FOR Ø-9)
140 BCS VALIT; IF NOT, EXIT THIS ROUTINE (WE'VE STORED THE NUMBER AND HAVE
150 STA HEXBUF,X; LOCATED SOMETHING OTHER THAN AN ASCII NUMBER)
160 INX; KEEP STORING VALID ASCII NUMBERS IN HEXBUF BUFFER
170 JMP MATH2;-------------------
180 RANGECK CMP #58;-------------- IS THIS >47 AND <58
190 BCS MATH3
200 SEC
210 SBC #48
220 SEC
230 SBC #208; IS IT > 47 & < 58
240 MATH3 RTS
250 VALIT LDA #Ø;---------- TURN IT FROM ASCII INTO A 2-BYTE INTEGER
260 STA HEXBUF,X; PUT ZERO AT END OF ASCII NUMBER (AS DELIMITER)
270 LDA #<HEXBUF; POINT "TEMP" POINTER TO ASCII NUMBER IN BUFFER
280 STA TEMP
290 LDA #>HEXBUF
```

```
300 STA TEMP+1
310 JSR VALDEC; ROUTINE WHICH TURNS ASCII NUMBER INTO INTEGER IN "RESULT"
320 LDA RESULT; MOVE RESULT TO TEMPORARY ADDITION VARIABLE, "ADDNUM"
330 STA ADDNUM
340 LDA RESULT+1
350 STA ADDNUM+1
360 RTS; RETURN TO CALLER
370 .FILE PRINTOPS
```

For the Atari version of Math, change line 370 to:

370 .FILE D:PRINTOPS, SRC

Program 7-2. Printops

```
10 ; "PRINTOPS"   PRINTS & POKES VALUES (BOTH OPCODES & ARGUMENTS)
20 FORMAT LDA PASS; ON PASS 2, IGNORE INCSA (RAISES PC) SINCE
30 BNE PRM; ON PASS 2, WE JSR TO POKEIT (IT GOES TO INCSA)
40 JSR INCSA; BUT ON PASS 1, WE DON'T PRINT OR POKE ANYTHING, WE JUST
50 RTS; RAISE THE PC AND RETURN --------------
60 PRM LDA SFLAG; SHOULD WE PRINT TO SCREEN
70 BEQ PRMX; IF NOT, SKIP THIS NEXT PART (PRINT TO SCREEN)
80 JSR CLRCHN; OTHERWISE, RESET NORMAL I/O CONDITION
90 LDX #1; (FILE #1 FOR INPUT, SCREEN FOR OUTPUT)
100 JSR CHKIN
110 LDX OP; LOAD THE OPCODE
120 JSR PRNTNUM; PRINT IT
130 JSR PRNTSPACE; PRINT A SPACE
140 PRMX LDX OP;----------- NOW POKE THE OPCODE INTO RAM/DISK MEMORY
150 JSR POKEIT
160 RTS;-------------------------------------
```

```
170 ; PRINT TWO BYTES (THE OPCODE AND A 1-BYTE ARGUMENT)-----------------
180 PRINT2 LDA PASS; ON PASS 2, WE SKIP INCSA (SEE LINE 20 ABOVE)
190 BNE P2M
200 JSR INCSA
210 RTS;--------------------
220 P2M LDA SFLAG; IF SCREEN PRINT FLAG IS DOWN, SKIP PRINTING TO SCREEN
230 BEQ P2MX
240 LDX RESULT; OTHERWISE PRINT THE LOW-BYTE OF "RESULT" (THE ARGUMENT)
250 JSR PRNTNUM
260 P2MX LDX RESULT; AND ALSO POKE THE LOW-BYTE TO RAM/DISK MEMORY
270 JMP POKEIT; A JMP TO POKEIT WILL RTS US BACK TO THE CALLER------------
280 ; PRINT THREE BYTES (THE OPCODE AND A 2-BYTE ARGUMENT)-----------------
290 PRINT3 LDA PASS; ON PASS 2, SKIP INCSA (SEE LINE 20 ABOVE)
300 BNE P3M
310 JSR INCSA; RAISE PC BY 2
320 JSR INCSA
330 RTS;--------------------
340 P3M LDA SFLAG; SHOULE WE PRINT TO SCREEN
350 BEQ P3MX
360 LDX RESULT; PRINT AND POKE LOW BYTE OF ARGUMENT
370 JSR PRNTNUM
380 P3MX LDX RESULT
390 JSR POKEIT
400 LDA SFLAG; SHOULD WE PRINT TO SCREEN
410 BEQ P3MXX
420 LDA HXFLAG; ARE WE PRINTING OPCODES AND ARGUMENTS IN HEX
430 BEQ P3MX2; IF SO, DON'T PRINT A SPACE HERE
440 JSR PRNTSPACE; OTHERWISE, PRINT A SPACE
450 P3MX2 LDX RESULT+1; PRINT AND POKE THE HIGH BYTE OF THE ARGUMENT
460 JSR PRNTNUM
```

```
470 P3MXX LDX RESULT+1
480 JMP POKEIT; AND A JUMP TO POKEIT WILL RTS US BACK TO CALLER
490 POKEIT STX WORK+1;-------------POKE IN A BYTE TO RAM/DISK--------------
500 LDA POKEFLAG; ARE WE SUPPOSED TO POKE TO RAM
510 BEQ DISP; IF NOT, SKIP IT
520 LDY #Ø; OTHERWISE, SEND THE BYTE TO RAM MEMORY AT CURRENT PC ADDRESS (SA)
530 TXA
540 STA (SA),Y;----------------
550 DISP LDA DISKFLAG; ARE WE SUPPOSED TO POKE TO A DISK OBJECT FILE
560 BEQ INCSA; IF NOT, SKIP IT
570 JSR CLRCHN; IF SO, ALERT FILE #2 (WRITE FILE ON DISK)
580 LDX #2
590 JSR CHKOUT
600 LDA WORK+1; PUT THE BYTE TO BE SENT TO DISK IN THE A REGISTER
610 JSR PRINT; PRINT (AFTER LINES 55Ø-57Ø ABOVE) PRINTS TO DISK FILE #2
620 JSR CLRCHN; RESTORE NORMAL I/O (PRINT TO SCREEN AND
630 LDX #1; READ FROM FILE #1
640 JSR CHKIN
650 INCSA CLC;------------------- RAISE THE PC COUNTER (SA) BY 1 --------
660 LDA #1
670 ADC SA
680 STA SA
690 LDA #Ø
700 ADC SA+1
710 STA SA+1
720 RTS
730 ;-----------------------  PRINTOUT ROUTINES (TO SCREEN) ----------
740 PRNTMESS LDY #Ø; PRINT A MESSAGE (ERRORS USUALLY) TO THE SCREEN
750 MESSLOOP LDA (TEMP),Y; THESE MESSAGES ARE DELIMITED BY Ø AND ARE POINTED
760 BEQ MESSDONE; TO BY THE VARIABLE "TEMP"
770 JSR PRINT
```

```
780 JSR PTP; AFTER PRINTING A CHARACTER TO SCREEN, CHECK TO SEE IF IT SHOULD
790 INY;                          ALSO BE PRINTED TO THE PRINTER
800 JMP MESSLOOP
810 MESSDONE RTS;---------------------------
820 PRNTSPACE LDA #32; PRINT A SPACE CHARACTER
830 JSR PRINT
840 JSR PTP; SEE IF IT SHOULD ALSO GO TO THE PRINTER
850 RTS;-----------------------------------------
860 PRNTNUM STX X; PRINT A NUMBER (LOW BYTE IN X, HIGH BYTE IN A)
870 LDA HXFLAG; IF WE'RE PRINTING IN HEX, NOT DECIMAL, THEN
880 BEQ PRNTNUMD; USE THE HEXPRINT SUBROUTINE.  OTHERWISE, GO TO PRNTNUMD
890 TXA
900 JSR HEXPRINT
910 JSR PTPNU; CHECK IF NUMBER SHOULD BE PRINTED TO PRINTER AS WELL
920 LDX X; RESTORE NUMBER IN X BEFORE
930 RTS; RETURNING TO CALLER------------------------
940 PRNTNUMD LDA #0; PRINT A DECIMAL NUMBER
950 JSR OUTNUM; BASIC'S LINE NUMBER PRINTOUT ROUTINE
960 JSR PTPNU; SHOULD WE ALSO PRINT IT TO PRINTER
970 LDX X; RESTORE VALUE IN X BEFORE
980 RTS; RETURNING TO THE CALLER ----------------------------------
990 PRNTSA LDA HXFLAG; PRINT THE SA (PC, PROGRAM COUNTER)
1000 BEQ PRNTSAD; IF NOT HEX PRINTOUT, THEN USE DECIMAL ROUTINE BELOW
1010 LDA SA+1; OTHERWISE, PRINT LOW AND HIGH BYTES OF SA (AS HEX)
1020 JSR HEXPRINT; HIGH BYTE 1ST
1030 LDA SA
1040 JSR HEXPRINT
1050 JSR PTPSA; SHOULD WE ALSO PRINT SA TO PRINTER
1060 RTS;----------------
1070 PRNTSAD LDX SA; PRINT SA (DECIMAL VERSION)
```

```
1080 LDA SA+1
1090 JSR OUTNUM
1100 JSR PTPSA; PRINT TO PRINTER, TOO
1110 RTS;-------------------------------------------------
1120 PRNTCR LDA #13;                      PRINT A CARRIAGE RETURN
1130 JSR PRINT
1140 JSR PTP; SHOULD WE DO IT ON THE PRINTER TOO
1150 RTS;------------------------------------------------
1160 PRNTLINE LDX LINEN;                  PRINT A SOURCE CODE LINE NUMBER
1170 LDA LINEN+1
1180 JSR OUTNUM; BASIC ROUTINE (LOW BYTE IN X, HIGH IN A)
1190 JSR PTPLI; SHOULD WE ALSO PRINT LINE NUMBER TO PRINTER
1200 RTS; ---------------------------------------------
1210 PRNTINPUT LDA #<LABEL;               PRINT CONTENTS OF MAIN INPUT
1220 STA TEMP;                            BUFFER ("LABEL")
1230 LDA #>LABEL; POINT "TEMP" TO THE BUFFER AND THEN
1240 STA TEMP+1
1250 JSR PRNTMESS; USE GENERAL MESSAGE PRINTING ROUTINE
1260 RTS
1270 ;------------------------   ERROR PRINTOUT PREPARATIONS
1280 ERRING LDA #7; RING BELL
1290 JSR PRINT
1300 LDA #18; TURN ON REVERSE PRINTING TO HIGHLIGHT ERROR
1310 JSR PRINT
1320 JSR PRNTINPUT; PRINT CONTENTS OF MAIN INPUT BUFFER
1330 LDA #13; PRINT A CARRIAGE RETURN
1340 JSR PRINT
1350 RTS
1360 ;------------------------------------- PRINTOUT (TO PRINTER)
1370 ;(PTP PRINTS A SINGLE CHARACTER TO THE PRINTER).
```

```
1380 PTP LDX PASS; ON PASS 1, DO NO PRINTING TO PRINTER
1390 BNE PTP1
1400 RTS
1410 PTP1 LDX PRINTFLAG; IF PRINTFLAG IS DOWN, DO NOTHING, RETURN TO CALLER
1420 BNE MPTP
1430 RTS;---------
1440 MPTP STA A; SAVE CONTENTS OF ACCUMULATOR
1450 JSR CLRCHN; ALERT PRINTER
1460 LDX #4
1470 JSR CHKOUT
1480 LDA A; RECOVER A
1490 JSR PRINT; PRINT TO PRINTER
1500 JSR CLRCHN; RESTORE NORMAL I/O
1510 LDX #1
1520 JSR CHKIN
1530 RETT LDA A; RECOVER A
1540 RTS; RETURN TO CALLER
1550 ;-------------------   NUMBERS TO PRINTER
1560 PTPNU LDX PASS; SAME LOGIC AS LINES 1350+ ABOVE
1570 BNE PTPN1
1580 RTS
1590 PTPN1 LDX PRINTFLAG
1600 BNE MPTPN
1610 RTS
1620 MPTPN JSR CLRCHN
1630 LDX #4
1640 JSR CHKOUT
1650 LDA HXFLAG; HEX OR DECIMAL MODE
1660 BEQ MPTPND
1670 LDA X
1680 JSR HEXPRINT
```

```
1690 JMP FINPTP
1700 MPTPND LDA #0
1710 LDX X
1720 JSR OUTNUM
1730 FINPTP JSR CLRCHN
1740 LDX #1
1750 JSR CHKIN
1760 RTS
1770 ;-----------------  SA TO PRINTER
1780 PTPSA LDX PASS; SAME LOGIC AS LINES 1350+ ABOVE
1790 BNE PTPS1
1800 RTS
1810 PTPS1 LDX PRINTFLAG
1820 BNE MPTPSA
1830 RTS
1840 MPTPSA JSR CLRCHN
1850 LDX #4
1860 JSR CHKOUT
1870 LDX HXFLAG; HEX OR DECIMAL PRINTOUT
1880 BEQ MPTPSAD
1890 LDA SA+1
1900 JSR HEXPRINT
1910 LDA SA
1920 JSR HEXPRINT
1930 JMP FINPTPSA
1940 MPTPSAD LDA SA+1
1950 LDX SA
1960 JSR OUTNUM
1970 FINPTPSA JSR CLRCHN
1980 LDX #1
1990 JSR CHKIN
```

```
2000 RTS
2010 ;------------------ LINE NUMBER TO PRINTER
2020 PTPLI LDX PASS; SAME LOGIC AS LINES 1350+ ABOVE
2030 BNE PTPL1
2040 RTS
2050 PTPL1 LDX PRINTFLAG
2060 BNE MPTPL
2070 RTS
2080 MPTPL JSR CLRCHN
2090 LDX #4
2100 JSR CHKOUT
2110 LDA LINEN+1
2120 LDX LINEN
2130 JSR OUTNUM
2140 JSR CLRCHN
2150 LDX #1
2160 JSR CHKIN
2170 RTS
2180 ;-----------------------------------  HEX NUMBER PRINTOUT
2190 ; PRINT THE NUMBER IN THE ACCUMULATOR AS A HEX DIGIT (AS ASCII CHARS.)
2200 HEXPRINT PHA; STORE NUMBER
2210 AND #$0F; CLEAR HIGH BITS (10101111 BECOMES 00001111, FOR EXAMPLE)
2220 TAY; NOW WE KNOW WHICH POSITION IN THE STRING OF HEX NUMBERS ("HEXA")
2230 LDA HEXA,Y; THIS NUMBER IS.  SO PULL IT OUT AS AN ASCII CHARACTER
2240 ; (HEXA LOOKS LIKE THIS: "0123456789ABCDEF")
2250 TAX; SAVE LOW-BITS VALUE INTO X
2260 PLA; PULL OUT THE ORIGINAL NUMBER, BUT THIS TIME
2270 LSR;SHIFT RIGHT 4 TIMES (MOVING THE 4 HIGH BITS INTO THE 4 LOW BITS AREA)
2280 LSR; (10101111 BECOMES 00001010, FOR EXAMPLE)
2290 LSR
2300 LSR
```

```
2310 TAY; AGAIN, PUT POSITION OF THIS VALUE INTO THE Y INDEX
2320 LDA HEXA,Y; PULL OUT THE RIGHT ASCII CHARACTER FROM "HEXA" STRING
2330 JSR PRINT; PRINT HIGH VALUE (FIRST) (A HOLDS HIGH VALUE AFTER LINE 2280)
2340 TXA; (X HELD LOW VALUE AFTER LINE 2210)
2350 JSR PRINT; PRINT LOW VALUE
2360 RTS; RETURN TO CALLER
2370 .FILE PSEUDO
```

Program 7-3. Printops, Atari Modifications

To create the Atari version of Printops, change the following lines in Program 7-2:

```
610 JSR OBJPRINT
1280 ERRING LDA #253
2370 .FILE D:PSEUDO.SRC
```

Chapter 8

Pseudo:

I/O and Linked Files

Pseudo: I/O and Linked Files

All pseudo-ops except .BYTE (and in-line ones like #< or +) are handled by the Pseudo subprogram. Eight pseudo-ops are tested for at the start of Pseudo (**50–300**). They are: .FILE, .END, .D, .P, .N, .O, .S, and .H. These tests and the associated JMPs are identical to an ON-GOTO multiple branch structure in a BASIC program. The rest of the Pseudo subprogram is a collection of subroutines which service these various pseudo-ops.

If an unrecognized pseudo-op appears within the source code, an error message is printed out (**340–460**). If something like .X or .MAP appears, the line number, the start address, and the source code line are printed (**350–390**). The variable TEMP is set to point to the SYNTAX ERROR message in the Tables subprogram, and that message is sent to screen, and possibly printer, via the PRNTMESS subroutine (**440**). A carriage return is printed (**450**), and we return to the Eval subprogram after pulling all the characters of the current source code line. The subroutine PULLINE does this (**460**).

Assuming, however, that LADS came upon the legitimate pseudo-op .FILE during an assembly, lines 480–830 take the necessary action. .FILE appears at the end of a subprogram. It tells LADS that another subprogram is linked to the one just assembled and that the source code within this next subprogram is to be assembled next, as an extension of the current subprogram. The current source code file will need to be shut down, and the next linked file will need to be opened for business. The next linked file is the one called NAME, for example, in .FILE NAME.

Linking with .FILE

The FILE subroutine starts off by looking for a blank character following the .FILE pseudo-op word (**480–510**). Locating a blank, it can now store the name of the next file of source code. It pulls in the name, one character at a time, looking for an end-of-line 0 (**540**) or a byte with the seventh bit set (a tokenized keyword which needs to be stretched out into a full ASCII word). Then each character in NAME is stored in the main buffer (**590**) as it comes in from the source code.

When an end-of-line 0 is encountered, the whole filename has been stored in LABEL, the input buffer. And—since Y was counting the number of characters and helping store them in the right place in the buffer—Y now holds the number of characters in the filename, its length. We store Y in the FNAMELEN variable which will be needed by the DOS (Disk Operating System) when the OPEN1 subroutine tries to open or load a program file on the disk.

Now the filename is moved from the LABEL buffer to the FILEN buffer (**630–680**). Why not just store the name in the FILEN buffer in the first place? First, because the printout routines get their characters and words from LABEL, the main buffer. Second, because there might be a keyword, a tokenized, abbreviated BASIC command within a filename. The filename might be END or IFNOT. And KEYWORD, our detokenization subroutine, acts upon words in LABEL, the main buffer. So, rather than make a separate KEYWORD detokenization subroutine for each buffer, it's easier to bring words into the main buffer first, detokenizing them on the fly. Then move them.

But why, then, not have the OPEN1 subroutine look to the main buffer for its filenames? That way, the names wouldn't need to be moved to FILEN, a separate buffer. True enough, but it helps me and, I suspect, many other programmers to keep things separated by function.

It takes only 14 bytes in LADS to move the filename from the main buffer to the filename buffer. It adds only a few microseconds during assembly time since .FILE is a relatively rare event. It won't happen more than a few times during an entire assembly. It's nowhere near the heavy action of the innermost loops of LADS where every event counts, where every improvement in logic results in a noticeable improvement in speed. So memory use or speed efficiency is not really worth bothering with here. If it's easier for you to visualize the actions of a program (and make sure there are no unwanted *interactions*), use as many buffers and variables as you want.

Printing Addresses

The next section of this FILE subroutine prints out to screen or printer (**690–740**). Pass 2 doesn't print the starting address of each linked file. That's one way to tell which pass is currently being assembled. Change the LDA PASS in line 690 to LDA

#0 if you want the address printed on both passes. The PRNTSA subroutine (from Printops) prints the address in RAM memory where the first byte in the new file will be assembled. PRNTINPUT prints the filename from the main buffer. Then a carriage return prepares for the next screen (or printer) line (**740**). The whole thing looks like this on the screen:

470A NAME
49FF NEXTNAME

If the .S and .P pseudo-ops are turned off, nothing will be printed to the screen during an assembly except for this list of linked files and their object code addresses. That's the fastest way to assemble any source code. Printing during assembly takes up a considerable amount of time.

The OPEN1 closes the old source code file and opens the new one. OPEN1 is found in the subprogram of the same name. Next, the computer's input channel is switched to file #1, the input-from-disk channel, and two bytes are pulled off the newly opened source code program file. (These first two bytes are, in the Commodore DOS system, ignorable.) Then ENDPRO gets us in position to analyze the first line in this new source code file (**800**). Finally, the ENDFLAG is set down because there's obviously more code to assemble. We return to line 80 where the RTS (back to the Indisk subprogram) is pulled off the stack, and we JMP directly back into the Eval subprogram to pull in the first source code line of the newly opened file.

The .END Link

The .END pseudo-op is quite like the .FILE pseudo-op. It serves to link the *last* file in a chain to the *first* file:

PROG1	(ends with .FILE PROG2)
PROG2	(ends with .FILE PROG3)
PROG3	(ends with .END PROG1, pointing back to the original file)

This way, the assembler can go through two passes.

.END starts off by printing the word .END (**850–940**). Then it borrows a good section of the FILE subroutine above the JSRing to line 520. Most of the events in FILE now take place: The name of the new program file is stored in the two buffers, the file is opened, ENDPRO puts us in the right spot to look for a new line, and so on. When we return to the END subroutine (**970**), .END's most important work is now performed: On pass 1, the ENDFLAG is left down (**980**). But on pass 2, the ENDFLAG is sent up, and that will quickly cause the Eval subprogram to shut the entire LADS engine down.

But if this is pass 1, another very important thing happens: *Pass 1 is changed into pass 2.* The PASS flag itself is set up (**1000**).

The original starting address is now retrieved from the TA variable and restored into SA, the main Program Counter variable. This starts us off on the second pass with the correct, original starting address for assembling the object code. The JSR to INDISK gets us pointed to the first true line of source code in that first program file (past the *= symbol), and we RTS back up to line 140 which exits us from this subprogram the same way that the .FILE pseudo-op exits.

Assembly to Disk Object File

The .DISK pseudo-op is an important one: It makes it possible to store the object code, the results of an assembly, as a program on disk. In a way, it's the opposite of .FILE. .FILE pulls in source code from a program file already on the disk; .DISK sends out object code to a new program file being actively created during the assembly process.

On pass 1, nothing is stored to a disk object file, so we branch to PULLJ which is a springboard to PULLINE. PULLINE pulls in the rest of a logical line and prepares us to look at the next logical line.

On pass 2, however, all object code is stored to a disk object file if the .D NAME pseudo-op has been invoked. This storage happens character by character, just the way that object code is sent to the screen or printer. But before these bytes can go into a disk object code file, the file must be opened for *writing* on disk.

One character is pulled off the source code, moving us past the space character in .D NAME and pointing to the N in NAME. A little loop (**1130–1210**) stores the NAME of the object file into the main buffer (for printouts) and into the filename buffer, FILEN, simultaneously. Meanwhile, if any tokenized keywords are detected (seventh bit set), we're directed to translate them to ASCII characters via a JSR KEYWORD (**1170**). This accomplished, we add ",P,W" onto the end of the filename.

That's Commodore-specific; it tells the DOS that this file is to be a Program/Write file.

At this point, Y holds the length of the filename, and it's then stored in the proper zero page location (**1350**) for use by the DOS in opening this write file. Now the main input line, the filename, is printed out, and the DISKFLAG is set up (**1380**). That tells LADS to always send object code bytes to this object file on pass 2 when it has finished assembling each logical line.

An Abnormal Program

The routine OPEN2 in the Open1 subprogram will now open the write file on disk (**1390**), and the *channel* to that file is made the main output channel at this point (**1400–1410**). Whatever is PRINTed will now go to the disk write file. And the first two bytes of a program file tell the computer where in RAM memory to load a program file. Normally, for a BASIC program, this load address would be the start of RAM, the start of BASIC's storage area for programs. But this is an abnormal program. It's machine language; it could go anywhere in RAM. We therefore need to tell the computer what the starting address of this particular program is.

At the very beginning of LADS, the start address is pulled from just beyond the source code's *= symbol. That symbol must be the first item in any source code. The start address is then put into several variables. SA, the Program Counter, gets it, but will keep raising it as each logical line is assembled. SA is a dynamic, changing variable. TA also gets the start address. TA is a "variable," but never changes. Its job is to remember the starting address all through the assembly process. Perhaps TA should be called a *constant* rather than a *variable*, but the term *variable* is generally used in computing to refer to both types of "remember this" storage places.

TA Remembers

In any event, TA will always know where we started assembling. So TA is sent to the disk object file as the first two bytes (**1420–1450**) and then normal I/O (input from disk source file, output to screen) is restored (**1460–1470**). Now a disk error is checked for, and we prepare to look at the next logical line via JSR ENDPRO (**1500**). The RTS is pulled off the stack (it would want to send us back to INDISK), we set the ENDFLAG down and JMP back to Eval to analyze the next line of source code (**1550**).

The PRINTER subroutine responds to a .P pseudo-op. It is ignored on pass 1, but on pass 2 the file to the printer is opened (**1590**), and the PRINTFLAG is raised. Normal I/O is restored, and we "fall through" to PULLINE, the subroutine which keeps sucking bytes off the current logical line until the end of that line is reached. These bytes are ignored. That's why pseudo-ops should be the only thing on any physical line. Anything following a pseudo-op is sucked in and ignored.

The PULLINE routine finishes when a colon or a 0 is detected. The exit back to STARTLINE in Eval is prepared for by the PLA PLA which throws away the RTS (caused by JSRing to Pseudo from within Indisk). The only difference between a 0 (end-of-physical-line) and a colon (end-of-logical-line) condition is that a 0 requires that we skip over some link bytes in the source code. 0 requires that we first clean off these link bytes by a JSR to ENDPRO (**1700**). ENDPRO is also necessary in the event that the end of a physical line is also the end of the source code file itself. ENDPRO would detect that.

The .O pseudo-op notifies LADS that you want object code stored into RAM memory during assembly beginning at the start address *=. This is relatively simple: We just print out the .O (**1770–1800**) and set up the POKEFLAG. (Elsewhere in LADS, the POKEFLAG is queried to determine if object code should be sent to RAM.) Then we exit via PULLINE.

Turning Things Off

The .N pseudo-op turns things off. It can turn four things off: printer printout, RAM object code storage, screen printout, and hexadecimal printout. If .N is detected in the ON-GOTO section of Pseudo above (**110–320**), we are sent here for another ON-GOTO series of tests (**1880–1960**). Of course, none of these forms of output are triggered on pass 1, so they don't need to be turned off on pass 1 either. But on pass 2, we are sent to one of the four turn-it-off routines below.

NIXPRINT (**1980**) first notifies us that the .NP pseudo-op has been detected in the source code by printing the .NP. Then the PRINTFLAG is lowered (**2050**), and a carriage return is sent to the printer. This is in case you should want the printer turned on again further along in the source code. (You would turn it on with the .P pseudo-op.) The first line of a reactivated printout must appear on a new line, not as an extension of the previous printout.

Then the printer is turned off with JSR CLOSE (this close-down-a-file routine is in the Open1 subprogram), and we exit via PULLINE (**2160**).

The next three turn-it-off pseudo-ops are simple, and virtually identical. NIXOP prints .NO and sets down the POKEFLAG. NIXHEX prints .NH and sets down the HXFLAG (causing decimal to become the number base for opcode printouts to printer and screen). NIXSCREEN prints .NS and sets down the SFLAG. Each routine exits via PULLINE described above.

Disk Error Trapping

DISERR (**2510**) checks for an error in disk operation. It could be JSRed to from any place in LADS where you suspect that things aren't likely to go well with the disk. Disk drives differ considerably in their reliability: An unabused Commodore 4040 drive is usually good for years of error-free performance; many of the Commodore 1541 single-drive units, especially the earlier ones, are perhaps best described as *sensitive*. In any case, how often you feel the need to JSR DISERR for a report on the disk's success in completing an operation will depend on how often your drive is the cause of problems during your other programming experience.

For Commodore computers, a simple check of the ST (status) byte in zero page will reveal many kinds of disk errors. If one is detected, an error message is printed and LADS is shut down (**2650**) by jumping to FIN within Eval.

The .S (screen printout on) and .H (hexadecimal number printout) pseudo-ops are the final items to assemble as part of the LADS source code program. The subprogram Table follows, but it's *data*, not programming.

There's no particular reason why these two pseudo-ops should be the last thing in LADS. They just are.

Also, they're very simple. They each print their names to announce themselves, .S or .H; set up their flags, SFLAG or HXFLAG; and exit through PULLINE. The only notable thing about .S is that it must not set its flag until pass 2.

The .H is a default condition of this assembler. LADS assumes that you want hex output unless you use the .NH to turn off hex and turn decimal on. Of course, you can set up other default conditions which are more harmonic with your own programming needs.

Program 8-1. Pseudo

```
10 ; "PSEUDO" HANDLE ALL PSEUDOPS EXCEPT .BYTE
15 ;
20 ; JMP HERE FROM INDISK
30 ; (INDISK WAS JSR'ED TO FROM EVAL). / Y HOLDS POINTER TO LABEL
40 ; -------------------------
50 PSEUDO CMP #70; IS IT "F" FOR .FILE
60 BNE PSE1
70 JSR FILE; F MEANS GO TO NEXT LINKED FILE -----------------
80 GOBACK PLA; RETURN TO EVAL TO GET NEXT LINE
90 PLA
100 JMP STARTLINE;------------------------
110 PSE1 CMP #128; IS IT .END
120 BNE PSEE
130 JSR PEND; 128 IS TOKEN FOR END (END OF CHAIN PSEUDO)
140 JMP GOBACK; RETURN TO EVAL
150 PSEE CMP #68; IS IT "D" FOR .DISK (CREATE OBJECT CODE FILE ON DISK)
160 BNE PSEE1
170 JMP PDISK; OPEN FILE ON DISK FOR OBJECT CODE STORAGE
180 PSEE1 CMP #80; IS IT "P" FOR .P (PRINTER OUTPUT)
190 BNE PSEE2
200 JMP PPRINTER; TURN ON PRINTER LISTING
210 PSEE2 CMP #78; IS IT "N" FOR .NH OR .NS OR SOME OTHER "TURN IT OFF"
220 BNE PSEE3
230 JMP NIX; TURN SOMETHING OFF
240 PSEE3 CMP #79; IS IT "O" FOR OUTPUT (POKE OBJECT CODE INTO RAM)
250 BNE PSEE4
260 JMP OPON; START POKING OBJECT CODE (DEFAULT)
270 PSEE4 CMP #83; IS IT "S" FOR PRINT TO SCREEN
280 BNE PSEE5
```

```
290 JMP SCREIN; TURN ON SCREEN PRINTING
300 PSEE5 CMP #72; IS IT "H" FOR HEX NUMBERS DURING PRINTOUTS
310 BNE PSE9;
320 JMP HEXIT; TURN ON HEX PRINTING
330 ;------------------------ PRINT ERROR MESSAGE (NO SUCH PSEUDO-OP)
340 PSE9 STA LABEL,Y; STORE CHAR. FOR PRINTOUT
350 JSR PRNTLINE
360 JSR PRNTSPACE
370 JSR PRNTSA
380 JSR ERRING
390 JSR PRNTINPUT
400 LDA #<MERROR
410 STA TEMP
420 LDA #>MERROR
430 STA TEMP+1
440 JSR PRNTMESS
450 JSR PRNTCR
460 JMP PULLINE; PULL IN (& IGNORE) REST OF LINE, THEN BACK TO EVAL
470 ;------------------------ HANDLE .FILE PSEUDO-OP ----------------
480 FILE JSR CHARIN
490 CMP #32; LOOK FOR END OF THE WORD .FILE (TO LOCATE FILENAME)
500 BEQ FIØ
510 JMP FILE; CONTINUE LOOKING FOR BLANK
520 FIØ LDY #Ø
530 FI1 JSR CHARIN
540 CMP #Ø; END OF LINE
550 BEQ FI2
560 CMP #127; KEYWORD, SO STRETCH IT OUT
570 BCC FII1
580 JSR KEYWORD
590 FII1 STA LABEL,Y; STORE CHAR. OF FILENAME
```

```
600 INY
610 JMP FI1; CONTINUE STORING FILENAME IN MAIN BUFFER (LABEL)
620 FI2 STY FNAMELEN; STORE FILENAME LENGTH
630 LDY #0
640 FILO LDA LABEL,Y;------- PUT FILENAME INTO PROPER BUFFER (FILEN)
650 BEQ FILO1
660 STA FILEN,Y
670 INY
680 JMP FILO
690 FILO1 LDA PASS; ON PASS 2, DON'T PRINT OUT PC
700 BNE FI5
710 JSR PRNTSA; PRINT THE FILENAME
720 JSR PRNTSPACE
730 FI5 JSR PRNTINPUT
740 JSR PRNTCR; CARRIAGE RETURN
750 JSR OPEN1; OPEN NEXT LINKED FILE ON DISK (FOR CONTINUED READING OF SOURCE)
760 LDX #1
770 JSR CHKIN
780 JSR CHARIN; PULL IN NEXT TWO BYTES AND
790 JSR CHARIN
800 JSR ENDPRO; CHECK FOR END OF PROGRAM
810 LDX #0
820 STX ENDFLAG; SET END OF PROGRAM FLAG TO ZERO
830 RTS
840 ;------------------------ HANDLE .END PSEUDO-OP --------
850 PEND LDA #46; PRINT OUT .END
860 JSR PRINT
870 LDA #69
880 JSR PRINT
890 LDA #78
900 JSR PRINT
```

```
910 LDA #68
920 JSR PRINT
930 LDA #32
940 JSR PRINT
950 JSR CHARIN
960 JSR FI0; GET FILENAME, ETC. JUST AS .FILE PSEUDO-OP DOES
970 LDA PASS; ON PASS 1, DON'T SET THE ENDFLAG UP.
980 BEQ PEND1; BUT ON PASS 2, IT'S NECESSARY (TO END THE ENTIRE PROGRAM)
990 INC ENDFLAG
1000 PEND1 INC PASS; RAISE PASS FROM PASS 1 TO PASS 2
1010 LDA TA; PUT ORIGINAL START ADDRESS BACK INTO PC (SA) FOR RESTART OF
1020 STA SA; ASSEMBLY ON PASS 2.
1030 LDA TA+1
1040 STA SA+1
1050 JSR INDISK; SET UP NEXT LINE
1060 RTS
1070 ;------------------- HANDLE .D FILENAME PSEUDO-OP (OBJECT CODE FILE)
1080 PDISK LDA PASS; ON PASS 1, DON'T STORE ANYTHING TO DISK
1090 BEQ PULLJ; PULLJ IS A SPRINGBOARD (JUMPS TO PULLINE)
1100 JSR CHARIN; POINT TO FILENAME
1110 STA LABEL,Y
1120 LDY #0
1130 PDLOOP JSR CHARIN
1140 BEQ PD1; END OF LINE
1150 CMP #127; IT'S A KEYWORD (WITHIN THE FILENAME) IF >127
1160 BCC PDIX
1170 JSR KEYWORD
1180 PDIX STA LABEL,Y; KEEP STORING FILENAME INTO PRINTOUT BUFFER (LABEL)
1190 STA FILEN,Y; AS WELL AS OPEN1 BUFFER (FILEN)
1200 INY
1210 JMP PDLOOP; KEEP STORING FILENAME;------------------
```

```
1220 PULLJ JMP PULLINE;-------- SPRINGBOARD TO IGNORE FILENAME
1230 PD1 LDA #44; PUT ,P,W (PROGRAM, WRITE) SIGNALS ONTO FILENAME
1240 STA FILEN,Y
1250 INY
1260 LDA #80
1270 STA FILEN,Y
1280 INY;  ADD--,P,W
1290 LDA #44
1300 STA FILEN,Y
1310 INY
1320 LDA #87
1330 STA FILEN,Y
1340 INY
1350 STY FNAMELEN; STORE FILENAME LENGTH
1360 JSR PRNTINPUT; PRINT OUT THE LINE
1370 JSR PRNTCR; CARRIAGE RETURN
1380 INC DISKFLAG; RAISE DISKFLAG TO SHOW THAT FUTURE POKES SHOULD GO TO DISK
1390 JSR OPEN2; OPEN A SECOND DISK FILE (THIS ONE FOR WRITING TO)
1400 LDX #2
1410 JSR CHKOUT
1420 LDA TA; PRINT OBJECT CODE'S STARTING ADDRESS TO DISK FILE
1430 JSR PRINT
1440 LDA TA+1
1450 JSR PRINT
1460 EDISK JSR CLRCHN
1470 LDX #1; RESTORE NORMAL I/O
1480 JSR CHKIN
1490 JSR DISERR; CHECK FOR DISK ERROR (FAILURE TO OPEN CORRECTLY)
1500 JSR ENDPRO; GET NEXT LINE NUMBER
1510 PLA; PULL RTS
1520 PLA
```

```
1530 LDX #0
1540 STX ENDFLAG; RESET END OF PROGRAM FLAG
1550 JMP STARTLINE; AND RETURN TO EVAL TO GET NEXT LINE
1560 ;------------------------- HANDLE .P (PRINTER) PSEUDO-OP -------
1570 PPRINTER LDA PASS; ON PASS 1, DO NO PRINTER OUTPUT
1580 BEQ PULLINE; GET RID OF REST OF LINE AND GO ON.
1590 JSR OPEN4; PASS 2, SO OPEN PRINTER TO HEAR FROM COMPUTER
1600 INC PRINTFLAG; RAISE PRINTER OUTPUT FLAG (SO PRINT WILL SEND BYTES TO
1610 JSR CLRCHN;                   THE PRINTER AS WELL AS THE SCREEN).
1620 LDX #1; RESTORE NORMAL I/O
1630 JSR CHKIN
1640 ;---------------------- SUCTION ROUTINE.  GET RID OF REST OF A LINE
1650 PULLINE JSR CHARIN; IGNORE ALL BYTES, JUST LOCATE NEXT LINE
1660 BEQ ENDPULL; ZERO END OF LINE SHOULD GO TO ENDPRO FOR NEXT LINE #
1670 CMP #58; WHEREAS A COLON END OF LINE SKIPS THAT STEP
1680 BEQ ENDPULR; (COLON)
1690 JMP PULLINE; NEITHER COLON NOR ZERO (SO PULL IN MORE CHARACTERS)
1700 ENDPULL JSR ENDPRO
1710 ENDPULR PLA; PULL RTS OFF STACK
1720 PLA
1730 LDX #0
1740 STX ENDFLAG; SET ENDFLAG DOWN
1750 JMP STARTLINE; RETURN TO EVAL (TO GET NEXT LINE OF SOURCE CODE)
1760 ;------------------------- HANDLE .O (POKE BYTES TO RAM) PSEUDO-OP
1770 OPON LDA #46; PRINT .O
1780 JSR PRINT
1790 LDA #79; "O"
1800 JSR PRINT
1810 JSR PRNTCR; CARRIAGE RETURN
1820 LDA #1
1830 STA POKEFLAG; RAISE POKE-TO-RAM FLAG
```

```
1840 JMP PULLINE; IGNORE REST OF LINE
1850 ;------------------------- HANDLE .N(SOMETHING),TURN-IT-OFF PSEUDO-OPS
1860 NIX LDA PASS; ON PASS 1, DON'T BOTHER WITH ANY OF THIS
1870 BEQ PULLINE
1880 JSR CHARIN; ON PASS 2, SEE WHICH THING IS BEING TURNED OFF
1890 CMP #80; IS IT ".NP" TO "NOT PRINT TO PRINTER"
1900 BEQ NIXPRINT
1910 CMP #79; IS IT ".NO" TO "NOT POKE OBJECT BYTES TO RAM"
1920 BEQ NIXOP
1930 CMP #83; IS IT ".NS" TO "NOT PRINT TO SCREEN"
1940 BEQ NIXSCREEN
1950 CMP #72; IS IT ".NH" TO "NOT PRINTOUT HEX" (THUS SWITCH TO DECIMAL)
1960 BEQ NIXHEX
1970 ;------------------------- TURN OFF PRINTER OUTPUT
1980 NIXPRINT LDA #46; PRINT ".NP" TO SCREEN
1990 JSR PRINT
2000 LDA #78; "N"
2010 JSR PRINT
2020 LDA #80; "P"
2030 JSR PRINT
2040 JSR PRNTCR; CARRIAGE RETURN
2050 DEC PRINTFLAG; LOWER PRINT-TO-SCREEN FLAG
2060 JSR CLRCHN; TURN OFF PRINTER
2070 LDX #4
2080 JSR CHKOUT
2090 LDA #13
2100 JSR PRINT
2110 LDA #4
2120 JSR CLOSE
2130 JSR CLRCHN
2140 LDX #1; RESTORE NORMAL I/O
```

```
2150 JSR CHKIN
2160 JMP PULLINE; IGNORE REST OF LINE (AND RETURN TO EVAL)
2170 ;------------------------- STOP POKEING OBJECT BYTES TO RAM
2180 NIXOP LDA #46; PRINT ".NO"
2190 JSR PRINT
2200 LDA #78;      "N"
2210 JSR PRINT
2220 LDA #79;      "O"
2230 JSR PRINT
2240 JSR PRNTCR;CARRIAGE RETURN
2250 LDA #0
2260 STA POKEFLAG; TURN OFF POKE FLAG
2270 JMP PULLINE; IGNORE REST OF LINE (AND RETURN TO EVAL)
2280 ;------------------------- STOP HEX PRINTOUTS (START DECIMAL)
2290 NIXHEX LDA #46; PRINT ".NH"
2300 JSR PRINT
2310 LDA #78;      "N"
2320 JSR PRINT
2330 LDA #72;      "H"
2340 JSR PRINT
2350 JSR PRNTCR; CARRIAGE RETURN
2360 LDA #0
2370 STA HXFLAG; PUT HEXFLAG DOWN
2380 JMP PULLINE; IGNORE REST OF LINE (AND RETURN TO EVAL)
2390 ;------------------------- STOP SCREEN PRINTOUTS
2400 NIXSCREEN LDA #46; PRINT ".NS"
2410 JSR PRINT
2420 LDA #78;      "N"
2430 JSR PRINT
2440 LDA #83;      "S"
2450 JSR PRINT
```

```
2460 JSR PRNTCR;CARRIAGE RETURN
2470 LDA #0
2480 STA SFLAG; PUT DOWN SCREEN PRINTOUT FLAG
2490 JMP PULLINE; IGNORE REST OF LINE (AND RETURN TO EVAL)
2500 ;------------------------  DISK ERROR DETECTION ROUTINE --------
2510 DISERR LDX ST; CHECK DISK STATUS VARIABLE (COMPUTER SPECIFIC)
2520 BNE MODIER; IF NOT ZERO, THERE IS SOME FAULT IN THE DISK I/O
2530 RTS;----------
2540 MODIER LDA #0; PRINT OUT ERROR MESSAGE
2550 JSR PRNTNUM
2560 JSR PRNTSPACE
2570 LDA #<MDISER
2580 STA TEMP; POINT TO DISK ERROR MESSAGE
2590 LDA #>MDISER
2600 STA TEMP+1
2610 JSR ERRING; RING BELL
2620 JSR PRNTMESS; PRINT ERROR MESSAGE
2630 PLA; PULL RTS OFF STACK
2640 PLA
2650 JMP FIN; SHUT DOWN ENTIRE LADS OPERATION
2660 ;--------------- HANDLE .S PSEUDO-OP (TURN ON SCREEN PRINTOUT)
2670 SCREIN LDA #46; PRINT ".S"
2680 JSR PRINT
2690 LDA #83;        "S"
2700 JSR PRINT
2710 JSR PRNTCR; CARRIAGE RETURN
2720 LDA PASS; ON PASS 1, NO SCREEN PRINTOUT
2730 BEQ SCREX
2740 LDA #1; OTHERWISE, RAISE SCREEN PRINTOUT (LISTING) FLAG
2750 STA SFLAG
2760 SCREX JMP PULLINE; IGNORE REST OF LINE (AND RETURN TO EVAL)
```

```
2770 ;-------------- HANDLE .H PSEUDO-OP (HEX NUMBERS DURING PRINTOUT)
2780 HEXIT LDA #46; PRINT ".H"
2790 JSR PRINT
2800 LDA #72;       "H"
2810 JSR PRINT
2820 JSR PRNTCR; CARRIAGE RETURN
2830 LDA #1
2840 STA HXFLAG; SET HEXFLAG UP
2850 JMP PULLINE; IGNORE REST OF LINE (AND RETURN TO EVAL)
2860 .FILE TABLES
```

Program 8-2. Pseudo, Apple Modifications

To create the Apple version of Pseudo, omit lines 1230–1340 and lines 2500–2650 from Program 8-1 and change the following lines:

```
110 PSE1 CMP #69; IS IT .END
960 JSR FILE; GET FILENAME, ETC. JUST AS .FILE PSEUDO-OP DOES
1002 SEC; SAVE LENGTH OF FILE
1003 LDA SA; FOR THIRD AND FOURTH
1004 SBC TA; BYTES OF BINARY FILE
1005 STA LENPTR; CREATED BY .D
1006 LDA SA+1; PSEUDO-OP
1007 SBC TA+1
1008 STA LENPTR+1
1350 PD1 STY FNAMELEN
1455 LDA LENPTR; WRITE LENGTH OF
1456 JSR PRINT; BINARY FILE
1457 LDA LENPTR+1
```

```
1458 JSR PRINT
1490 ;
```

Program 8-3. Pseudo, Atari Modifications

To create the Atari version of Pseudo, omit lines 1230–1340 and lines 1400–1450 from Program 8-1 and change the following lines:

```
10 ;ATARI MODIFICATIONS--PSEUDO
110 PSE1 CMP #69
580 :
675 CPY FNAMELEN
680 BNE FILO
780 :
790 :
800 :
960 JSR FILE
1001 LDA SA
1002 STA LLSA
1003 LDA SA+1
1004 STA LL3A+1
2510 DISERR LDX $0363
2530 .FILE D:KERNAL.SRC
```

Chapter 9

Tables:

Data, Messages, Variables

Tables:
Data, Messages, Variables

Computers are information processors. *Data* is another word for information. This points up the difference between the two distinct sections of any computer program: code and data. The code, or program proper, is a list of actions for the computer to take. The data is the information upon which those actions are based.

Data is usually separated from the code; it might even be outside the computer. Sometimes data is on a disk file, sometimes on tape, sometimes in the user's brain, as when a program halts and asks for input from a keyboard. In all of these cases, though, the code is segregated from the data which it processes.

An Odd Duck

LADS processes source code, turning it into runnable object code. It takes a list of actions like LDA #75:STA SCREEN and turns them into computer-understandable machine language object programs.

LADS gets its data from two sources, a disk source code file (or source code in RAM) and also from the Tables subprogram. Tables isn't really a sub*program*, of course. We're forced to call it that because there isn't a better word. It's really an odd duck. There are no commands to the computer within Tables. It's pure information. Essential information, true, but there are no ML instructions in Tables. Just definitions, messages, pointers, buffers, flags, and registers. LADS couldn't operate without them, but they're not active programming instructions—they're for reference.

Three Parallel Tables

Tables starts out, appropriately enough, with three parallel tables: MNEMONICS, TYPES, and OPS. Each table contains 56 pieces of information. MNEMONICS holds the names of all the 6502 mnemonics like LDA and INY. TYPES identifies the *category* of each mnemonic (we'll get to this in a minute). And OPS provides an opcode for each category. To see how these three tables work together, let's look at the first item in the first table, the mnemonic LDA.

In your machine language programming, you might want to load the Accumulator with the number 1. You would write:

100 LDA #1

The computer wouldn't grasp the meaning of the ASCII characters L-D-A-#-1 at all. They're for our convenience, not its.

We think alphabetically or alphanumerically. It thinks binarily. It wants pure numbers. The CPU, the "thinking" part of the 6502 chip, takes action according to a code of its own, but this code isn't the ASCII code. It's an *opcode*, an operations code. The CPU will place a number into the Accumulator, the A Register, if it comes across any of the following numbers: 161, 165, 169, 173, 177, 181, 185, or 189. Each of these numbers is an opcode for LDA. But each one loads from a different place. The different numbers represent the opcodes for the eight different *addressing modes* available to LDA. They are:

Addressing Mode's Name	Example	Opcode
Immediate	LDA #15	169
Zero Page	LDA 15	165
Zero Page,X	LDA 15,X	181
Zero Page,X (indirect)	LDA (15,X)	161
Zero Page,Y (indirect)	LDA (15),Y	177
Absolute	LDA 1500	173
Absolute,Y	LDA 1500,Y	185
Absolute,X	LDA 1500,X	189

Most of the mnemonics can use a variety of addressing modes. LDA can be addressed these eight ways, LDY can be addressed five ways, and so on. That's where TYPES comes in. There are ten TYPES, and each opcode falls into one of the ten categories. Mnemonics are grouped according to their addressing mode's similarities. The mnemonics cluster into TYPES according to the way that they can be addressed:

Type 0:

RTS, INY, DEY, DEX, INX, SEC,
CLC, TAX, TAY, TXA, TYA, PHA,
PLA, BRK, CLD, CLI, PHP, PLP,
RTI, SED, SEI, TSX, TXS, CLV
NOP

(Each of these mnemonics takes up only one byte in memory; each is only capable of *Implied* addressing—they have no argument, no address.)

Type 1:
LDA, CMP, STA, SBC, ADC, AND,
ORA, EOR

(Type 1 mnemonics have the largest number of possible addressing modes, eight. See the list for LDA above.)

Type 2:
STY, STX, DEC, INC

(These are fairly restricted in their addressing options. STY has only three possibilities: Absolute, Zero Page, and Zero Page,X. STX can perform only Absolute, Zero Page, and Zero Page,Y [it's the only one which can use this Zero Y mode]. DEC and INC can do Absolute; Zero Page; Zero Page,X; and Absolute,X.)

Type 3:
ROL, ROR, LSR, ASL

(These are the bit-shifting, "logical" instructions. They can be addressed in the following modes: Absolute; Zero Page; Zero Page,X; Absolute,X; and one which is reserved for them alone, Accumulator mode. In that mode, the number held in the Accumulator is acted upon.)

Type 4:
CPY, CPX

(The compare X or Y can use Immediate, Absolute, or Zero Page modes.)

Type 5:
LDY, LDX

(These loads are more restricted in their addressing possibilities than LDA. LDX can use Immediate; Absolute; Zero Page; Absolute,Y; and Zero Page,Y. LDY can use Immediate; Absolute; Zero Page; Zero Page,X; and Absolute,X. Notice that they cannot index themselves; ,X modes are possible only with LDY and vice versa.)

Type 6:
JMP

(This is a special case; it stands alone. It has two ways of addressing: the extremely common Absolute mode and the extremely rare Indirect mode, JMP (via this). Because most programming contains many JMPs, it should have its own category. Also, the only other mnemonic which is essentially limited to Absolute addressing is JSR, and it gets a category all to itself as well.)

Type 7:
BIT

(This one is also an oddity. It too needs a category all its own. BIT can use only Absolute or Zero Page addressing.)

Type 8:
BCS, BEQ, BCC, BNE, BMI, BPL,
BVC, BVS

(All the branch instructions collect together as type 8. They have only one addressing mode, Relative, and they are the only instructions which can use this mode.)

Type 9:
JSR

(It can only Absolute address.)

Each of these groups derives from the arrangement of the opcodes. The patterns are more easily visualized if you look at the opcodes laid out in a table according to their numeric values.

Table 9-1. Table of Opcodes

LSD / MSD	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	LSD / MSD
0	BRK	ORA IND,X				ORA Z Page	ASL Z Page		PHP	ORA IMM	ASL A			ORA ABS	ASL ABS		0
1	BPL	ORA IND,Y				ORA Z Page, X	ASL Z Page, X		CLC	ORA ABS,Y				ORA ABS,X	ASL ABS,X		1
2	JSR	AND IND,X			Bit Z Page	AND Z Page	ROL Z Page		PLP	AND IMM	ROL A		BIT ABS	AND ABS	ROL ABS		2
3	BMI	AND IND,Y				AND Z Page,X	ROL Z Page,X		SEC	AND ABS,Y				AND ABS,X	ROL ABS,X		3
4	RTI	EOR IND,X				EOR Z Page	LSR Z Page		PHA	EOR IMM	LSR A		JMP ABS	EOR ABS	LSR ABS		4
5	BVC	EOR IND,Y				EOR Z Page,X	LSR Z Page,X		CLI	EOR ABS Y				EOR ABS,X	LSR ABS,X		5
6	RTS	ADC IND,X				ADC Z Page	ROR Z Page		PLA	ADC IMM	ROR A		JMP IND	ADC ABS	ROR ABS		6
7	BVS	ADC IND,Y				ADC Z Page,X	ROR Z Page, X		SEI	ADC ABS,Y				ADC ABS,X	ROR ABS,X		7
8		STA,IND X			STY Z Page	STA Z Page	STX Z Page		DEY		TXA		STY ABS	STA ABS	STX,ABS		8
9	BCC	STA IND,Y			STY Z Page,X	STA Z Page,X	STX Z Page,Y		TYA	STA ABS,Y	TXS			STA ABS,X			9
A	LDY IMM	LDA IND,X	LDX IMM		LDY Z Page	LDA Z Page	LDX Z Page		TAY	LDA IMM	TAX		LDY ABS	LDA ABS	LDX ABS		A
B	BCS	LDA IND,Y			LDY Z Page,X	LDA Z Page,X	LDX Z Page,Y		CLV	LDA ABS,Y	TSX		LDY ABS,X	LDA ABS,X	LDX ABS,Y		B
C	CPY IMM	CMP IND,X			CPY Z Page	CMP Z Page	DEC Z Page		INY	CMP IMM	DEX		CPY ABS	CMP ABS	DEC ABS		C
D	BNE	CMP IND,Y				CMP Z Page,X	DEC Z Page,X		CLD	CMP ABS,Y				CMP ABS,X	DEC ABS,X		D
E	CPX IMM	SBC IND,X			CPX Z Page	SBC Z Page	INC Z Page		INX	SBC IMM	NOP		CPX ABS	SBC ABS	INC ABS		E
F	BEG	SBC IND,Y				SBC Z Page,X	INC Z Page,X		SED	SBC ABS,Y				SBC ABS,X	INC ABS,X		F

Notice the relationship between LDA (15,X) and LDA #15. The former has an opcode of 161; the latter, 169. As the Eval subprogram goes through the source code line, it is looking for clues to the addressing mode: Is there a #, a comma, a parenthesis, an X, or a Y?

Each of these things, combined with the TYPE, tells Eval when to raise the value of the original opcode (let's call it the *base opcode)* assigned to the mnemonic from the OPS table. If Eval finds a # symbol, it adds 8 to the base opcode and goes right to the TWOS exit. It knows then that this opcode should be 169 (161 + 8) and that there will be *two bytes* to assemble: Immediate mode addressing uses two bytes. (All the other mnemonics grouped with LDA as type 1 will also add 8 to their base opcodes to signify their Immediate addressing modes.)

The base opcodes are in that third table called OPS (**190**). The Eval subprogram looks up each mnemonic in the MNEMONICS table, and then the numbers extracted from the TYPES and OPS tables are stored in the variables TYPE and OP for future reference. Finally, Eval starts looking for those # and) clues within the source code line. These clues cause Eval to add 4 or 8 or 16 or sometimes even 24 to the base opcode. This adjusts the base opcode upward so it will eventually become the correct opcode for the addressing mode being used.

CMP is grouped with LDA as a type 1 mnemonic. That's because a # will add 8 to either of their base opcodes and result in the correct, final opcode for Immediate addressing. The base opcode for CMP is 193, which, unadjusted, would stand for CMP (15,X). If we come upon a # following the CMP, however, 8 is added to the 193, giving 201, the correct opcode for CMP #15. Then Eval would JMP to TWOS and conclude assembly of that line of source code.

In each case, the base opcode in the OPS table is the lowest possible opcode number from among the addressing mode options available to each mnemonic. As the evaluation process proceeds throughout the Eval subprogram, the discovery of the various addressing modes triggers additions to the base opcode. In the end, when Eval finally releases a source code line, the right opcode has been achieved.

Returning to the data within the Tables subprogram, we next come upon the little HEXA table (**270**). It lists all the digits found in hexadecimal numbers. It's used as a lookup table when LADS translates an internal two-byte integer into a printable, readable ASCII hexadecimal number like F-F-D-2.

The Six Bufferettes

Here are the buffers (**290–340**). They are constantly being filled with a source code line, evaluated, and then cleaned off by being filled with zeros. They are separated into six different bufferettes primarily for the programmer's benefit. It's easier to visualize different actions if the buffers have different names.

LABEL is the main buffer—every source code line comes into it. BUFFER is where arguments are sent for further study. The rest of them are used for special-purpose analysis. Things like hex numbers are moved up to HEXBUF, for example, so they will be isolated from other characters and can be translated.

One other buffer, distant from the rest, is needed. LADS stores comments (remarks following semicolons in the source code) into a buffer normally used by BASIC to hold program lines. The location of this buffer depends on each computer's memory organization and so it is defined in the Defs subprogram.

The computer's Accumulator and Y and X are called *registers*. They're like hypervariables inside the 6502 chip—they are constantly changing. Calling them registers serves to distinguish them from program-created variables or other special locations within the computer. The three variables RADD, VREND, and TSTORE are called registers in LADS. That's largely the result of whimsy. There are as yet no established conventions concerning how to describe storage areas in ML programming. In this book we're variously referring to these set-aside bytes as flags, variables, registers, pointers, vectors, etc. (See Chapter 1).

In reality, they're all pretty much the same thing: Just some RAM memory space we've allocated with the .BYTE pseudo-op (or identified in zero page by definition using the = pseudo-op like STATUS = $FD). But it's nice to use various terms. It helps to remember things and, sometimes, it even helps to describe the purpose or function of a particular variable. *Pointers*, for example, are always associated with the Indirect Y addressing mode—LDA (POINTER),Y. They point to some address in RAM.

Registers Used by Valdec

Anyway, these three variables are described (**350**) as registers. RADD holds numbers being added to other numbers. VREND holds the length of the ASCII version of a number while it's being turned into an integer. TSTORE holds the interim results of multiplication. All three "registers" are used by the Valdec subprogram.

Lines 400–460 contain the various error messages. Note that each one ends with .BYTE 0 to stick a delimiting 0 in after the message itself. This 0 tells PRNTMESS (the subroutine in the Printops subprogram which prints messages) where to stop.

The rest of Tables contains variables, pointers, and registers. Notice that there are no zero page variables here. Zero page variables, pointers especially, are most useful for Indirect Y addressing, but you won't need too many of them. In fact, you won't be allowed to use much of zero page because it is so popular with your computer's operating systems and languages. But the most important thing to remember about any zero page space that you do use is: *Zero page variables must be defined at the start of your assembler source code.* They are unique in this. Any other equates can be defined anywhere in the source code. And, of course, the address-type PC variables or labels can be defined anywhere.

OP and TYPE are variables which hold information about the mnemonic currently under investigation during assembly. After a mnemonic is located in the MNEMONIC table, the matching TYPE and base opcode are pulled out of their tables and stored into the variables OP and TP for later reference (**480–490**). TA is the permanent storage area for the start address of assembly, the original *=.

Source Code Line Numbers

LINEN holds the source code line number of whatever physical line is currently being assembled. ENDFLAG tells Eval when to shut down assembly. It is incremented by the .END pseudo-op. WORK is used by several routines within LADS as a convenient place to temporarily leave two-byte values.

RESULT is an important variable. It holds the argument of each opcode. When an argument (expression-type) label like STA HERE is encountered, the label HERE is looked up by the subprogram Array and the integer value of the word HERE is placed into RESULT. When a hex argument like STA $1500 comes in from the source code, the subprogram Indisk translates the characters *$1500* into an integer value and stores that value in RESULT. Likewise, a decimal argument like STA 5376 is sent to RESULT after it's evaluated in the Eval subprogram. For every addressing mode which has an argument, the argument is stored in RESULT after it's been evaluated.

ARGSIZE holds the length of each argument, how many characters long it is. For example, ARGSIZE would hold a 7 for the argument in LDA (155),Y since (155),Y is seven characters long. It is used in the Eval subprogram in lines 1670, 2250, 2750, and 3020.

EXPRESSF is a flag which shows whether or not there is a label being used as an argument. LDA 15 would leave EXPRESSF down. LDA NAME would set it up. It is used in the Eval subprogram at lines 740, 1470, 1510, 1590, and 1700.

HEXFLAG tells the Eval subprogram whether or not it must calculate a decimal argument. Hex arguments are calculated (and left in RESULT) by the Indisk subprogram. Decimal arguments, however, need to be worked out by Eval. HEXFLAG is used in lines 550 and 1680 in Eval.

HEXLEN holds the length of a hex number. It is used in Indisk in lines 2170, 2240, and 2490.

KEYNUM holds the position of a keyword (a BASIC command) in the table of keywords in ROM BASIC. It is used in Indisk in 1060, 1080, 4260, and 4280.

LABSIZE is used in the Equate subprogram to hold the number of characters in an equate-type label (such as NAME = 22). It is used in lines 120, 160, and 410.

LABPTR is also used by Equate. It points to the position in the label array where the integer value of a label should be stored. It is found in lines 600 and 750.

ARRAYTOP points to the highest byte in the label array. It is where we start any search through the labels. Identical to TA, ARRAYTOP also represents the start of the LADS assembler in memory, minus one. It is used in Equate in lines 110 and 150 and in Array in lines 30 and 50.

A List of Flags and Variables

BUFLAG goes up when a line of source code contains # or (. These symbols are important when determining addressing mode, but must be ignored in evaluating arguments (the numeric value of the expression). This flag is used in lines 470 and 1020 in Array and in lines 750 and 1400 in Eval.

PASS is used frequently throughout the entire LADS program—it shows which pass we're currently on during assembly. A 0 in PASS signifies pass 1; a 1 represents pass 2.

The three variables A, X, and Y are often called upon to temporarily hold the values in the 6502 registers after which they were named. They are temporary storage areas.

PT is a temporary storage area to hold the PARRAY dynamic pointer in the Array subprogram.

BNUMFLAG and BFLAG are used in the evaluation of the .BYTE pseudo-op in the Indisk subprogram.

ADDNUM holds the value of the number following the + pseudo-op. For example, it would hold 78 if this were the source code: LDA LABEL+78.

The PLUSFLAG shows that there is something in the ADDNUM variable which must be added to the label in an argument. It shows that the + pseudo-op appears in the current source code line.

BYTEFLAG shows that the < or > pseudo-op appears in the current source code line. It is an odd flag in that it has more than two states. It can be 0 indicating no < or >. And it can be 1 or 2 to distinguish between < and >.

DISKFLAG means the .D NAME pseudo-op was activated and so object code should be sent to a disk object file to create a runnable ML program.

PRINTFLAG means the .P pseudo-op was activated and a listing should go to the printer for a hard copy record of assembly.

POKEFLAG means the .O pseudo-op was activated and all object code generated by assembly should be POKEd into RAM memory.

COLFLAG is used in the Indisk subprogram to show that the previously assembled line of source code ended with a colon rather than a 0 (end of physical line). It tells Indisk not to look for a new source code line number.

FOUNDFLAG goes up when the same word is found more than once within the label array, proving that a label has been redefined. That's illegal and results in an error message. This flag is used in the Array subprogram.

SFLAG means the .S pseudo-op is being used and a visible listing of source and object code should appear on the screen during assembly.

HXFLAG responds to the .H pseudo-op. If set (that's the default, the normal start-up condition in LADS), all opcodes and arguments are printed (to screen or printer) in hexadecimal. HXFLAG is turned off by the .NH (no hex) pseudo-op and causes opcodes and arguments to be printed as decimal numbers.

LOCFLAG, when set, tells the printout routines within the Eval subprogram that they need to print a PC address-type label. For example, a line like:

100 START LDA #GREEN

requires special handling so that the address-type label START will be printed on screen or printer in the correct format (or that it will be printed at all). LOCFLAG is used in Eval in lines 790, 1210, and 3510.

BABFLAG shows that there is a semicolon on a line of source code. It signifies that a REMark, a comment, appears on this line. It tells the printout routines that there is a comment which must also be printed on the screen or the printer following the printout of the business part of a line.

Program 9-1. Tables

```
10 ; "TABLES"
15 ;
20 ;  TABLE OF MNEMONICS AND PARALLEL TABLE OF OPCODE/ADDRESS TYPE DATA
30 ;  BUFFERS AND MESSAGES, FLAGS, POINTERS, REGISTERS
40 ;-------------------------  MNEMONICS, TYPES, ADDRESS MODE OPCODES
50 MNEMONICS .BYTE "LDALDYJSRRTSBCSBEQBCCCMP
60 .BYTE "BNELDXJMPSTASTYSTXINYDEY
70 .BYTE "DEXDECINXINCCPYCPXSBCSEC
80 .BYTE "ADCCLCTAXTAYTXATYAPHAPLA
90 .BYTE "BRKBMIBPLANDORAEORBITBVC
100 .BYTE "BVSROLRORLSRCLDCLIASLPHP
110 .BYTE "PLPRTISEDSEITSXTXSCLVNOP
120 TYPES .BYTE 1 5 9 0 8 8 8 1
130 .BYTE 8 5 6 1 2 2 0 0
140 .BYTE 0 2 0 2 4 4 1 0
150 .BYTE 1 0 0 0 0 0 0 0
160 .BYTE 0 8 8 1 1 1 7 8
170 .BYTE 8 3 3 3 0 0 3 0
180 .BYTE 0 0 0 0 0 0 0 0
190 OPS .BYTE 161 160 32 96 176 240 144 193
200 .BYTE 208 162 76 129 132 134 200 136
210 .BYTE 202 198 232 230 192 224 225 56
220 .BYTE 97 24 170 168 138 152 72 104
230 .BYTE 0 48 16 33 1 65 36 80
240 .BYTE 112 34 98 66 216 88 2 8
250 .BYTE 40 64 248 120 186 154 184 234
260 ;---------------------  HEX ROUTINE TABLE  ------------------
270 HEXA .BYTE "0123456789ABCDEF"
280 ;---------------------  BUFFERS  -----------------------------
```

```
290 LABEL .BYTE Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø
300 BUFFER .BYTE Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø
310 BUFM .BYTE Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø
320 HEXBUF .BYTE Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø
330 FILEN .BYTE Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø
340 NUBUF .BYTE Ø Ø Ø Ø Ø Ø Ø
350 ;-----  REGISTERS USED BY VALDEC -------
360 RADD .BYTE Ø Ø;TEMPORARY REGISTER FOR DOUBLE ADDITION
370 VREND .BYTE Ø; TEMP REG TO HOLD END OF PROGRAM COUNTER
380 TSTORE .BYTE Ø Ø; TEMPORARY REGISTER FOR MULTIPLY
390 ;---- MESSAGES TO PRINT TO SCREEN ---------------------------
400 MNOSTART .BYTE "NO START ADDRESS":.BYTE Ø
410 MBOR .BYTE "-------------------- BRANCH OUT OF RANGE":.BYTE Ø
420 NOLAB .BYTE "UNDEFINED LABEL":.BYTE Ø
430 NOARG .BYTE "           NAKED LABEL":.BYTE Ø
440 MDISER .BYTE "      <<<<<<<< DISK ERROR >>>>>>>> ":.BYTE Ø
450 MDUPLAB .BYTE "      -- DUPLICATED LABEL -- ":.BYTE Ø
460 MERROR .BYTE "      -- SYNTAX ERROR -- ":.BYTE Ø
470 ;-----------  FLAGS, POINTERS, REGISTERS  ---------------------
480 OP .BYTE Ø;          OPCODE
490 TP .BYTE Ø;          TYPE
500 TA .BYTE Ø Ø;          START ADDRESS
510 LINEN .BYTE Ø Ø;       CURRENT LINE #
520 ENDFLAG .BYTE Ø;     END-OF-PROG FLAG
530 WORK .BYTE Ø Ø;        TEMP WORK AREA
540 RESULT .BYTE Ø Ø;      TEMP ANSWER AREA
550 ARGN .BYTE Ø Ø;        VALUE OF ARGUMENT
560 ARGSIZE .BYTE Ø;     LENGTH OF ARGUMENT
570 EXPRESSF .BYTE Ø;    IS IT AN EXPRESS LABEL
580 HEXFLAG .BYTE Ø;     HEX NUMBER FLAG
590 HEXLEN .BYTE Ø;      LENGTH OF HEX NUMBER
```

```
600 NUMSIZE .BYTE Ø;     LENGTH OF ASCII NUMBER IN BUFFER (FOR VALDEC)
610 KEYNUM .BYTE Ø;      POSITION OF KEYWORD IN BASIC'S TABLE
620 LABSIZE .BYTE Ø;     SIZE OF LABEL (EQUATE TYPE)
630 LABPTR .BYTE Ø Ø;      POINTS TO ARRAY POSITION FOR ARG STORAGE
640 ARRAYTOP .BYTE Ø Ø;    TOP OF ARRAYS--SAME AS MEMTOP BEFORE LABELS.
650 BUFLAG .BYTE Ø;      AVOID # OR ( DURING ARRAYS ANALYSIS
660 PASS .BYTE Ø;        WHICH PASS WE'RE ON.
670 A .BYTE Ø:X .BYTE Ø:Y .BYTE Ø; TO HOLD REGISTERS DURING P SUBR. CHECKER
680 PT .BYTE Ø Ø;          TEMPORARILY HOLDS PARRAY (IN "ARRAY") 2-BYTE
690 BNUMFLAG .BYTE Ø;    FOR .BYTE IN "INDISK"
700 BFLAG .BYTE Ø Ø;       FOR NUMWERK IN "INDISK"
710 ADDNUM .BYTE Ø Ø;      NUMBER TO ADD FOR + PSEUDO
720 PLUSFLAG .BYTE Ø;    FLAG SHOWS THAT + PSEUDO HAPPENED.
730 BYTFLAG .BYTE Ø;     SHOWS THAT < OR > HAPPENED.
740 DISKFLAG .BYTE Ø;    SHOWS TO SEND BYTES TO DISK OBJECT FILE
750 PRINTFLAG .BYTE Ø;   SHOWS TO SEND BYTES TO PRINTER
760 POKEFLAG .BYTE Ø;    SHOWS TO SEND BYTES TO MEMORY (OBJECT CODE)
770 COLFLAG .BYTE Ø;     ENCOUNTERED A COLON (USED BY INDISK)
780 FOUNDFLAG .BYTE Ø;   DUPLICATED LABEL NAME (USED BY ARRAY)
790 SFLAG .BYTE Ø;       SHOWS TO SEND SOURCECODE TO SCREEN
800 HXFLAG .BYTE Ø;      SHOWS TO PRINT SA AND OPCODES IN HEX
810 LOCFLAG .BYTE Ø;     SHOWS TO PRINT A PC ADDRESS LABEL
820 BABFLAG .BYTE Ø;     SHOWS TO PRINT A REM AFTER PRNTINPUT IN EVAL
830 ;----------------------
840 ; NOW LINK UP WITH 1ST FILE ("DEFS") TO PERMIT 2ND PASS.
850 ;
860 .END DEFS
```

Program 9-2. Tables, Apple Modifications

To create the Apple version of Tables, make the following changes and additions to Program 9-1:

```
295 .BYTE Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø
     Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø
305 .BYTE Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø
     Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø
315 .BYTE Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø
     Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø
      Ø Ø Ø Ø Ø Ø Ø
83Ø LENPTR .BYTE Ø Ø;    HOLDS LENGTH OF BINARY PROGRAM
84Ø FOPEN1 .BYTE Ø;      HOLDS THE CURRENT INPUT FILE
85Ø FOPEN2 .BYTE Ø;      HOLDS THE CURRENT OUTPUT FILE
855 ;------------ DOS-MANAGER CONTROL BYTES -------------------
86Ø OPNREAD .BYTE 1 Ø 1 Ø Ø 1 6 2
87Ø .BYTE 45 147 Ø Ø Ø 147 Ø 146 Ø Ø
88Ø OPNWRIT .BYTE 1 Ø 1 Ø Ø 1 6 4
89Ø .BYTE 128 149 Ø Ø 83 149 83 148 Ø Ø
9ØØ RD1B .BYTE 3 1 Ø Ø Ø Ø Ø Ø Ø Ø Ø
91Ø .BYTE Ø 147 Ø 146 Ø 145
92Ø WR1B .BYTE 4 1 Ø Ø Ø Ø Ø Ø
93Ø WRDATA .BYTE Ø Ø Ø Ø 83 149 83 148 83 147
94Ø CLOSER .BYTE 2 Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø 147 Ø 146 Ø 145
95Ø CLOSEW .BYTE 2 Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø 83 149 83 148 83 147
96Ø OPNI .BYTE Ø;        HOLDS THE FILE # OF THE CURRENT INPUT DEVICE
97Ø OPNO .BYTE Ø;        HOLDS THE FILE # OF THE CURRENT OUTPUT DEVICE
98Ø A1 .BYTE Ø;          TEMP STORAGE OF ACC
99Ø Y1 .BYTE Ø;          TEMP STORAGE OF Y-REG
1ØØØ ;--------------------
1Ø1Ø .END DEFS
```

Program 9-3. Tables, Atari Modifications

To create the Atari version of Tables, make the following changes and additions to Program 9-1:

```
10 :ATARI MODIFICATIONS--TABLES
825 LLSA .BYTE 0 0
860 .END D:DEFS.SRC
```

Chapter 10

6502 Instruction Set

6502 Instruction Set

Here are the 56 mnemonics, the 56 instructions you can give the 6502 (or 6510) chip. Each of them is described in several ways: what it does, what major uses it has in ML programming, what addressing modes it can use, what flags it affects, its opcode (hex/decimal), and the number of bytes it uses up.

ADC

What it does: Adds byte in memory to the byte in the Accumulator, plus the carry flag if set. Sets the carry flag if result exceeds 255. The result is left in the Accumulator.

Major uses: Adds two numbers together. If the carry flag is set prior to an ADC, the resulting number will be *one* greater than the total of the two numbers being added (the carry is added to the result). Thus, one always clears the carry (CLC) before beginning any addition operation. Following an ADC, a set (up) carry flag indicates that the result exceeded one byte's capacity (was greater than 255), so you can chain-add bytes by subsequent ADCs without any further CLCs (see "Multi-Byte Addition" in Appendix D).

Other flags affected by addition include the V (overflow) flag. This flag is rarely of any interest to the programmer. It merely indicates that a result became larger than could be held within bits 0–6. In other words, the result "overflowed" into bit 7, the highest bit in a byte. Of greater importance is the fact that the Z is set if the result of an addition is zero. Also the N flag is set if bit 7 is set. This N flag is called the "negative" flag because you can manipulate bytes thinking of the seventh bit as a sign (+ or −) to accomplish "signed arithmetic" if you want to. In this mode, each byte can hold a maximum value of 127 (since the seventh bit is used to reveal the number's sign). The B branching instruction's Relative addressing mode uses this kind of arithmetic.

ADC can be used following an SED which puts the 6502 into "decimal mode." Here's an example. Note that the number 75 is *decimal* after you SED:

SED	
CLC	
LDA #75	
ADC #$05	(this will result in 80)
CLD	(always get rid of decimal mode as soon as you've finished)

Attractive as it sounds, the decimal mode isn't of much real value to the programmer. LADS will let you work in decimal if you want to without requiring that you enter the 6502's mode. Just leave off the $ and LADS will handle the decimal numbers for you.

Addressing Modes:

Name	**Format**	**Opcode**	**Number of Bytes Used**
Immediate	ADC #15	$69/105	2
Zero Page	ADC 15	$65/101	2
Zero Page,X	ADC 15,X	$75/117	2
Absolute	ADC 1500	$6D/109	3
Absolute,X	ADC 1500,X	$7D/125	3
Absolute,Y	ADC 1500,Y	$79/121	3
Indirect,X	ADC (15,X)	$61/97	2
Indirect,Y	ADC (15),Y	$71/113	2

Affected flags: N Z C V

AND

What it does: Logical ANDs the byte in memory with the byte in the Accumulator. The result is left in the Accumulator. All bits in both bytes are compared, and if both bits are 1, the result is 1. If either or both bits are 0, the result is 0.

Major uses: Most of the time, AND is used to turn bits off. Let's say that you are pulling in numbers higher than 128 (10000000 and higher) and you want to "unshift" them and print them as lowercase letters. You can then put a zero into the seventh bit of your "mask" and then AND the mask with the number being unshifted:

LDA ?	(test number)
AND #$7F	(01111111)

(If *either* bit is 0, the result will be 0. So the seventh bit of the test number is turned off here and all the other bits in the test number are unaffected.)

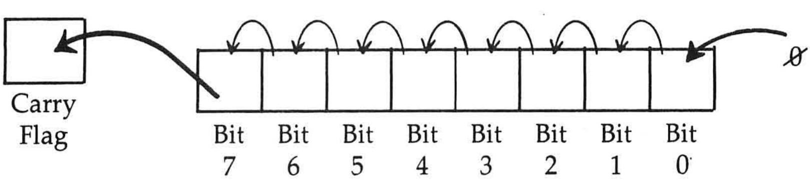
Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Immediate	AND #15	$29/41	2
Zero Page	AND 15	$25/37	2
Zero Page,X	AND 15/X	$35/53	2
Absolute	AND 1500	$2D/45	3
Absolute,X	AND 1500,X	$3D/61	3
Absolute,Y	AND 1500,Y	$39/57	3
Indirect,X	AND (15,X)	$21/33	2
Indirect,Y	AND (15),Y	$31/49	2

Affected flags: N Z

ASL

What it does: Shifts the bits in a byte to the left by 1. This byte can be in the Accumulator or in memory, depending on the addressing mode. The shift moves the seventh bit into the carry flag and shoves a 0 into the zeroth bit.



Major uses: Allows you to multiply a number by 2. Numbers bigger than 255 can be manipulated using ASL with ROL (see "Multiplication" in Appendix D).

A secondary use is to move the lower four bits in a byte (a four-bit unit is often called a *nybble*) into the higher four bits. The lower bits are replaced by zeros, since ASL stuffs zeros into the zeroth bit of a byte. You move the lower to the higher nybble of a byte by: ASL ASL ASL ASL.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Accumulator	ASL	$0A/10	1
Zero Page	ASL 15	$06/6	2
Zero Page,X	ASL 15,X	$16/22	2
Absolute	ASL 1500	$0E/14	3
Absolute,X	ASL 1500,X	$1E/30	3

Affected flags: N Z C

BCC

What it does: Branches up to 127 bytes forward or 128 bytes backward from its own address if the carry flag is clear. In effect, it branches if the second item is lower than the first, as in: LDA #150: CMP #149 or LDA #22: SBC #15. These actions would clear the carry and, triggering BCC, a branch would take place.

Major uses: For testing the results of CMP or ADC or other operations which affect the carry flag. IF-THEN or ON-GOTO type structures in ML can involve the BCC test. It is similar to BASIC's > instruction.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Relative	BCC addr.	$90/144	2

Affected flags: none of them.

BCS

What it does: Branches up to 127 bytes forward or 128 bytes backward from its own address if the carry flag is set. In effect, it branches if the second item is higher than the first, as in: LDA #150: CMP #249 or LDA #22: SBC #85. These actions would set the carry and, triggering BCS, a branch would take place.

Major uses: For testing the results of LDA or ADC or other operations which affect the carry flag. IF-THEN or ON-

GOTO type structures in ML can involve the BCC test. It is similar to BASIC's < instruction.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Relative	BCS addr.	$B0/176	2

Affected flags: none of them.

BEQ

What it does: Branches up to 127 bytes forward or 128 bytes backward from its own address if the zero flag (Z) is set. In other words, it branches if an action on two bytes results in a 0, as in: LDA #150: CMP #150 or LDA #22: SBC #22. These actions would set the zero flag, so the branch would take place.

Major uses: For testing the results of LDA or ADC or other operations which affect the carry flag. IF-THEN or ON-GOTO type structures in ML can involve the BEQ test. It is similar to BASIC's = instruction.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Relative	BEQ addr.	$F0/240	2

Affected flags: none of them.

BIT

What it does: Tests the bits in the byte in memory against the bits in the byte held in the Accumulator. The bytes (memory and Accumulator) are unaffected. BIT merely sets flags. The Z flag is set as if an Accumulator AND memory had been performed. The V flag and the N flag receive *copies* of the sixth and seventh bits of the tested number.

Major uses: Although BIT has the advantage of not having any effect on the tested numbers, it is infrequently used because you cannot employ the Immediate addressing mode with it. Other tests (CMP and AND, for example) can be used instead.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Zero Page	BIT 15	$24/36	2
Absolute	BIT 1500	$2C/44	3

Affected flags: N Z V

BMI

What it does: Branches up to 127 bytes forward or 128 bytes backward from its own address if the negative (N) flag is set. In effect, it branches if the seventh bit has been set by the most recent event: LDA #150 or LDA #128 would set the seventh bit. These actions would set the N flag, signifying that a *minus number* is present if you are using signed arithmetic or that there is a *shifted character* (or a BASIC keyword) if you are thinking of a byte in terms of the ASCII code.

Major uses: Testing for BASIC keywords, shifted ASCII, or graphics symbols. Testing for + or − in signed arithmetic.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Relative	BMI addr.	$30/48	2

Affected flags: none of them.

BNE

What it does: Branches up to 127 bytes forward or 128 bytes backward from its own address if the zero flag is clear. In other words, it branches if the result of the most recent event is not zero, as in: LDA #150: SBC #120 or LDA #128: CMP #125. These actions would clear the Z flag, signifying that a result was not 0.

Major uses: The reverse of BEQ. BNE means Branch if Not Equal. Since a CMP subtracts one number from another to perform its comparison, a 0 result means that they are equal. Any other result will trigger a BNE (not equal). Like the other B branch instructions, it has uses in IF-THEN, ON-GOTO type structures and is used as a way to exit loops (for example, BNE will branch back to the start of a loop until a 0 delimiter is encountered at the end of a text message). BNE is like BASIC's <> instruction.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Relative	BNE addr.	$D0/208	2

Affected flags: none of them.

BPL

What it does: Branches up to 127 bytes forward or 128 bytes backward from its own address if the N flag is clear. In effect, it branches if the seventh bit is clear in the most recent event, as in: LDA #12 or LDA #127. These actions would clear the N flag, signifying that a *plus number* (or zero) is present in signed arithmetic mode.

Major uses: For testing the results of LDA or ADC or other operations which affect the negative (N) flag. IF-THEN or ON-GOTO type structures in ML can involve the BCC test. It is the opposite of the BMI instruction. BPL can be used for tests of "unshifted" ASCII characters and other bytes which have the seventh bit off and so are lower than 128 (0XXXXXXX).

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Relative	BPL addr.	$10/16	2

Affected flags: none of them.

BRK

What it does: Causes a forced interrupt. This interrupt cannot be masked (prevented) by setting the I (interrupt) flag within the Status Register. If there is a Break Interrupt Vector (a vector is like a pointer) in the computer, it may point to a resident monitor if the computer has one. The PC and the Status Register are saved on the stack. The PC points to the location of the BRK + 2.

Major uses: Debugging an ML program can often start with a sprinkling of BRKs into suspicious locations within the code. The ML is executed, a BRK stops execution and drops you into the monitor, you examine registers or tables or variables to see if they are as they should be at this point in the execution, and then you restart execution from the breakpoint. This instruction is essentially identical to the actions and uses of the STOP command in BASIC.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Implied	BRK	$00/0	1

Affected flags: Break (B) flag is set.

BVC

What it does: Branches up to 127 bytes forward or 128 bytes backward from its own address if the V (overflow) flag is clear.

Major uses: None. In practice, few programmers use "signed" arithmetic where the seventh bit is devoted to indicating a positive or negative number (a set seventh bit means a negative number). The V flag has the job of notifying you when you've added, say 120 + 30, and have therefore set the seventh bit via an "overflow" (a result greater than 127). The result of your addition of two positive numbers should not be seen as a negative number, but the seventh bit *is* set. The V flag can be tested and will then reveal that your answer is still positive, but an overflow took place.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Relative	BVC addr.	$50/80	2

Affected flags: none of them.

BVS

What it does: Branches up to 127 bytes forward or 128 bytes backward from its own address if the V (overflow) flag is set).

Major uses: None. See BVC above.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Relative	BVS addr.	$70/112	2

Affected flags: none of them.

CLC

What it does: Clears the carry flag. (Puts a 0 into it.)

Major uses: Always used before any addition (ADC). If there are to be a series of additions (multiple-byte addition), only the first ADC is preceded by CLC since the carry feature is necessary. There might be a carry, and the result will be incorrect if it is not taken into account.

The 6502 does not offer an addition instruction without the carry feature. Thus, you must always clear it before the first ADC so a carry won't be accidentally added.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Implied	CLC	$18/24	1

Affected flags: Carry (C) flag is set to zero.

CLD

What it does: Clears the decimal mode flag. (Puts a 0 into it.)

Major uses: Commodore computers execute a CLD when first turned on as well as upon entry to monitor modes (PET/CBM models) and when the SYS command occurs. Apple and Atari, however, can arrive in an ML environment with the D flag in an indeterminant state. An attempt to execute

ML with this flag set would cause disaster—all mathematics would be performed in "decimal mode." It is therefore suggested that owners of Apple and Atari computers CLD during the early phase, the initialization phase, of their programs. Though this is an unlikely bug, it would be a difficult one to recognize should it occur.

For further detail about the 6502's decimal mode, see SED below.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Implied	CLD	$D8/216	1

Affected flags: Decimal (D) flag is set to zero.

CLI

What it does: Clears the interrupt-disable flag. All interrupts will therefore be serviced (including maskable ones).

Major uses: To restore normal interrupt routine processing following a temporary suspension of interrupts for the purpose of redirecting the interrupt vector. For more detail, see SEI below.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Implied	CLI	$58/88	1

Affected flags: Interrupt (I) flag is set to zero.

CLV

What it does: Clears the overflow flag. (Puts a 0 into it.)

Major uses: None. (See BVC above.)

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Implied	CLV	$B8/184	1

Affected flags: Overflow (V) flag is set to zero.

CMP

What it does: Compares the byte in memory to the byte in the Accumulator. Three flags are affected, but the bytes in memory and in the Accumulator are undisturbed. A CMP is actually a subtraction of the byte in memory from the byte in the Accumulator. Therefore, if you LDA #15:CMP #15—the result (of the subtraction) will be zero, and BEQ would be triggered since the CMP would have set the Z flag.

Major uses: This is an important instruction in ML. It is central to IF-THEN and ON-GOTO type structures. In combination with the B branching instructions like BEQ, CMP allows the 6502 chip to make decisions, to take alternative pathways depending on comparisons. CMP throws the N, Z, or C flags up or down. Then a B instruction can branch, depending on the condition of a flag.

Often, an action will affect flags by itself, and a CMP will not be necessary. For example, LDA #15 will put a 0 into the N flag (seventh bit not set) and will put a 0 into the Z flag (the result was not 0). LDA does not affect the C flag. In any event, you could LDA #15: BPL TARGET, and the branch would take effect. However, if you LDA $20 and need to know if the byte loaded is *precisely* $0D, you must CMP #$0D:BEQ TARGET. So, while CMP is sometimes not absolutely necessary, it will never hurt to include it prior to branching.

Another important branch decision is based on > or < situations. In this case, you use BCC and BCS to test the C (carry) flag. And you've got to keep in mind the *order* of the numbers being compared. The memory byte is compared to the byte sitting in the Accumulator. The structure is: memory *is less than or equal to* the Accumulator (BCC is triggered because the carry flag was cleared). Or memory *is more than* Accumulator (BCS is triggered because the carry flag was set). Here's an example. If you want to find out if the number in the Accumulator is less than $40, just CMP #$41:BCC

LESSTHAN (be sure to remember that the carry flag is cleared if a number is less than *or equal*; that's why we test for less than $40 by comparing with a $41):

LDA #75
CMP #$41; IS IT LESS THAN $40?
BCC LESSTHAN

One final comment about the useful BCC/BCS tests following CMP: It's easy to remember that BCC means *less than or equal* and BCS means *more than* if you notice that C is less than S in the alphabet.

The other flag affected by CMPs is the N flag. Its uses are limited since it merely reports the status of the seventh bit; BPL triggers if that bit is clear, BMI triggers if it's set. However, that seventh bit does show whether the number is greater than (or equal to) or less than 128, and you can apply this information to the ASCII code or to look for BASIC keywords or to search data bases (BPL and BMI are used by LADS' data base search routines in the Array subprogram). Nevertheless, since LDA and many other instructions affect the N flag, you can often directly BPL or BMI without any need to CMP first.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Immediate	CMP #15	$C9/201	2
Zero Page	CMP 15	$C5/197	2
Zero Page,X	CMP 15,X	$D5/213	2
Absolute	CMP 1500	$CD/205	3
Absolute,X	CMP 1500,X	$DD/221	3
Absolute,Y	CMP 1500,Y	$D9/217	3
Indirect,X	CMP (15,X)	$C1/193	2
Indirect,Y	CMP (15),Y	$D1/209	2

Affected flags: N Z C

CPX

What it does: Compares the byte in memory to the byte in the X Register. Three flags are affected, but the bytes in memory and in the X Register are undisturbed. A CPX is actually a subtraction of the byte in memory from the byte in the X Register. Therefore, if you LDA #15:CPX #15—the result (of the subtraction) will be zero and BEQ would be triggered since the CPX would have set the Z flag.

Major uses: X is generally used as an index, a counter within loops. Though the Y Register is often preferred as an index since it can serve for the very useful Indirect Y addressing mode (LDA (15),Y)—the X Register is nevertheless pressed into service when more than one index is necessary or when Y is busy with other tasks.

In any case, the flags, conditions, and purposes of CPX are quite similar to CMP (the equivalent comparison instruction for the Accumulator). For further information on the various possible comparisons (greater than, equal, less than, not equal), see CMP above.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Immediate	CPX #15	$E0/224	2
Zero Page	CPX 15	$E4/228	2
Absolute	CPX 1500	$EC/236	3

Affected flags: N Z C

CPY

What it does: Compares the byte in memory to the byte in the Y Register. Three flags are affected, but the bytes in memory and in the Y Register are undisturbed. A CPX is actually a subtraction of the byte in memory from the byte in the Y Register. Therefore, if you LDA #15: CPY #15—the result (of the subtraction) will be zero, and BEQ would be triggered since the CPY would have set the Z flag.

Major uses: Y is the most popular index, the most heavily used counter within loops since it can serve two purposes: It permits the very useful Indirect Y addressing mode (LDA (15),Y) and can simultaneously maintain a count of loop events.

See CMP above for a detailed discussion of the various branch comparisons which CPY can implement.

to reverse the current state of the sixth bit in a given byte: LDA BYTE:EOR #$40:STA BYTE. This will set bit 6 in BYTE if it was 0 (and clear it if it was 1). This selective bit toggling could be used to "shift" an unshifted ASCII character via EOR #$80 (1000000). Or if the character were shifted, EOR #$80 would make it lowercase. EOR toggles.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Immediate	EOR #15	$49/73	2
Zero Page	EOR 15	$45/69	2
Zero Page,X	EOR 15,X	$55/85	2
Absolute	EOR 1500	$4D/77	3
Absolute,X	EOR 1500,X	$5D/93	3
Absolute,Y	EOR 1500,Y	$59/89	3
Indirect,X	EOR (15,X)	$41/65	2
Indirect,Y	EOR (15),Y	$51/81	2

Affected flags: N Z

INC

What it does: Increases the value of a byte in memory by 1.

Major uses: Used exactly as DEC (see DEC above), except it counts up instead of down. For raising address pointers or supplementing the X and Y Registers as loop indexes.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Zero Page	INC 15	$E6/230	2
Zero Page,X	INC 15,X	$F6/246	2
Absolute	INC 1500	$EE/238	3
Absolute,X	INC 1500,X	$FE/254	3

Affected flags: N Z

INX

What it does: Increases the X Register by 1.

the X Register. Therefore, if you LDA #15:CPX #15—the result (of the subtraction) will be zero and BEQ would be triggered since the CPX would have set the Z flag.

Major uses: X is generally used as an index, a counter within loops. Though the Y Register is often preferred as an index since it can serve for the very useful Indirect Y addressing mode (LDA (15),Y)—the X Register is nevertheless pressed into service when more than one index is necessary or when Y is busy with other tasks.

In any case, the flags, conditions, and purposes of CPX are quite similar to CMP (the equivalent comparison instruction for the Accumulator). For further information on the various possible comparisons (greater than, equal, less than, not equal), see CMP above.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Immediate	CPX #15	$E0/224	2
Zero Page	CPX 15	$E4/228	2
Absolute	CPX 1500	$EC/236	3

Affected flags: N Z C

CPY

What it does: Compares the byte in memory to the byte in the Y Register. Three flags are affected, but the bytes in memory and in the Y Register are undisturbed. A CPX is actually a subtraction of the byte in memory from the byte in the Y Register. Therefore, if you LDA #15: CPY #15—the result (of the subtraction) will be zero, and BEQ would be triggered since the CPY would have set the Z flag.

Major uses: Y is the most popular index, the most heavily used counter within loops since it can serve two purposes: It permits the very useful Indirect Y addressing mode (LDA (15),Y) and can simultaneously maintain a count of loop events.

See CMP above for a detailed discussion of the various branch comparisons which CPY can implement.

to reverse the current state of the sixth bit in a given byte: LDA BYTE:EOR #$40:STA BYTE. This will set bit 6 in BYTE if it was 0 (and clear it if it was 1). This selective bit toggling could be used to "shift" an unshifted ASCII character via EOR #$80 (1000000). Or if the character were shifted, EOR #$80 would make it lowercase. EOR toggles.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Immediate	EOR #15	$49/73	2
Zero Page	EOR 15	$45/69	2
Zero Page,X	EOR 15,X	$55/85	2
Absolute	EOR 1500	$4D/77	3
Absolute,X	EOR 1500,X	$5D/93	3
Absolute,Y	EOR 1500,Y	$59/89	3
Indirect,X	EOR (15,X)	$41/65	2
Indirect,Y	EOR (15),Y	$51/81	2

Affected flags: N Z

INC

What it does: Increases the value of a byte in memory by 1.

Major uses: Used exactly as DEC (see DEC above), except it counts up instead of down. For raising address pointers or supplementing the X and Y Registers as loop indexes.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Zero Page	INC 15	$E6/230	2
Zero Page,X	INC 15,X	$F6/246	2
Absolute	INC 1500	$EE/238	3
Absolute,X	INC 1500,X	$FE/254	3

Affected flags: N Z

INX

What it does: Increases the X Register by 1.

Major uses: Used exactly as DEX (see DEX above), except it counts up instead of down. For loop indexing.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Implied	INX	$E8/232	1

Affected flags: N Z

INY

What it does: Increases the Y Register by 1.

Major uses: Used exactly as DEY (see DEY above), except it counts up instead of down. For loop indexing and working with the Indirect Y addressing mode (LDA (15),Y).

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Implied	INY	$C8/200	1

Affected flags: N Z

JMP

What it does: Jumps to any location in memory.

Major uses: Branching long range. It is the equivalent of BASIC's GOTO instruction. The bytes in the Program Counter are replaced with the address (the argument) following the JMP instruction and, therefore, program execution continues from this new address.

Indirect jumping—JMP (1500)—is not recommended, although some programmers find it useful. It allows you to set up a table of jump targets and bounce off them indirectly. For example, if you had placed the numbers $00 $04 in addresses $88 and $89, a JMP ($0088) instruction would send the program to whatever ML routine was located in address $0400. Unfortunately, if you should locate one of your pointers on the edge of a *page* (for example, $00FF or $17FF), this Indirect JMP addressing mode reveals its great weakness. There is a bug which causes the jump to travel to the wrong place—JMP

($00FF) picks up the first byte of the pointer from $00FF, but the second byte of the pointer will be incorrectly taken from $0000. With JMP ($17FF), the second byte of the pointer would come from what's in address $1700.

Since there is this bug, and since there are no compelling reasons to set up JMP tables, you might want to forget you ever heard of Indirect jumping.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Absolute	JMP 1500	$4C/76	3
Indirect	JMP (1500)	$6C/108	3

Affected flags: none of them.

JSR

What it does: Jumps to a subroutine anywhere in memory. Saves the PC (Program Counter) address, plus three, of the JSR instruction by pushing it onto the stack. The next RTS in the program will then pull that address off the stack and return to the instruction following the JSR.

Major uses: As the direct equivalent of BASIC's GOSUB command, JSR is heavily used in ML programming to send control to a subroutine and then (via RTS) to return and pick up where you left off. The larger and more sophisticated a program becomes, the more often JSR will be invoked. In LADS, whenever something is printed to screen or printer, you'll often see a chain of JSRs performing necessary tasks: JSR PRNTCR: JSR PRNTSA:JSR PRNTSPACE:JSR PRNTNUM:JSR PRNTSPACE. This JSR chain prints a carriage return, the current assembly address, a space, a number, and another space.

Another thing you might notice in LADS and other ML programs is a PLA:PLA pair. Since JSR stuffs the correct return address onto the stack before leaving for a subroutine, you need to do something about that return address if you later decide *not to RTS* back to the position of the JSR in the program. This might be the case if you *usually* want to RTS, but in some particular cases, you don't. For those cases, you can take control of program flow by removing the return address from the stack (PLA:PLA will clean off the two-byte address) and then performing a direct JMP to wherever you want to go.

If you JMP out of a subroutine without PLA:PLA, you could easily overflow the stack and crash the program.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Absolute	JSR 1500	$20/32	3

Affected flags: none of them.

LDA

What it does: Loads the Accumulator with a byte from memory. *Copy* might be a better word than *load,* since the byte in memory is unaffected by the transfer.

Major uses: The busiest place in the computer. Bytes coming in from disk, tape, or keyboard all flow through the Accumulator, as do bytes on their way to screen or peripherals. Also, because the Accumulator differs in some important ways from the X and Y Registers, the Accumulator is used by ML programmers in a different way from the other registers.

Since INY/DEY and INX/DEX make those registers useful as counters for loops (the Accumulator couldn't be conveniently employed as an index; there is no INA instruction), the Accumulator is the main temporary storage register for bytes during their manipulation in an ML program. ML programming, in fact, can be defined as essentially the rapid, organized maneuvering of single bytes in memory. And it is the Accumulator where these bytes often briefly rest before being sent elsewhere.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Immediate	LDA #15	$A9/169	2
Zero Page	LDA 15	$A5/165	2
Zero Page,X	LDA 15,X	$B5/181	2
Absolute	LDA 1500	$AD/173	3
Absolute,X	LDA 1500,X	$BD/189	3
Absolute,Y	LDA 1500,Y	$B9/185	3
Indirect,X	LDA (15,X)	$A1/161	2
Indirect,Y	LDA (15),Y	$B1/177	2

Affected flags: N Z

LDX

What it does: Loads the X Register with a byte from memory.

Major uses: The X Register can perform many of the tasks that the Accumulator performs, but it is generally used as an index for loops. In preparation for its role as an index, LDX puts a value into the register.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Immediate	LDX #15	$A2/162	2
Zero Page	LDX 15	$A6/166	2
Zero Page,Y	LDX 15,Y	$B6/182	2
Absolute	LDX 1500	$AE/174	3
Absolute,Y	LDX 1500,Y	$BE/190	3

Affected flags: N Z

LDY

What it does: Loads the Y Register with a byte from memory.

Major uses: The Y Register can perform many of the tasks that the Accumulator performs, but it is generally used as an index for loops. In preparation for its role as an index, LDY puts a value into the register.

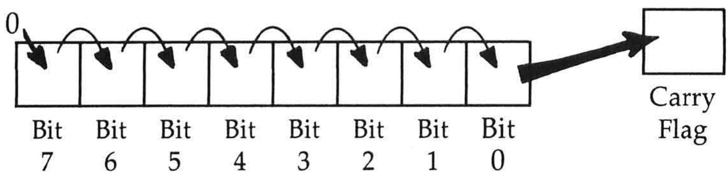
Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Immediate	LDY #15	$A0/160	2
Zero Page	LDY 15	$A4/164	2
Zero Page,X	LDY 15,X	$B4/180	2
Absolute	LDY 1500	$AC/172	3
Absolute,X	LDY 1500,X	$BC/188	3

Affected flags: N Z

LSR

What it does: Shifts the bits in the Accumulator or in a byte in memory to the right, by one bit. A zero is stuffed into bit 7, and bit 0 is put into the carry flag.



Major uses: To divide a byte by 2. In combination with the ROR instruction, LSR can divide a two-byte or larger number (see Appendix D).

LSR:LSR:LSR:LSR will put the high four bits (the high nybble) into the low nybble (with the high nybble replaced by the zeros being stuffed into the seventh bit and then shifted to the right).

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Accumulator	LSR	$4A/74	2
Zero Page	LSR 15	$46/70	2
Zero Page,X	LSR 15,X	$56/86	2
Absolute	LSR 1500	$4E/78	3
Absolute,X	LSR 1500,X	$5E/94	3

Affected flags: N Z C

NOP

What it does: Nothing. No operation.

Major uses: Debugging. When setting breakpoints with BRK, you will often discover that a breakpoint, when examined, passes the test. That is, there is nothing wrong at that place in the program. So, to allow the program to execute to the next breakpoint, you cover the BRK with a NOP. Then, when you run the program, the computer will slide over the NOP with no effect on the program. Three NOPs could cover a JSR XXXX, and you could see the effect on the program when that particular JSR is eliminated.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Implied	NOP	$EA/234	1

Affected flags: none of them.

ORA

What it does: Logically ORs a byte in memory with the byte in the Accumulator. The result is in the Accumulator. An OR results in a 1 if either the bit in memory or the bit in the Accumulator is 1.

Major uses: Like an AND mask which turns bits off, ORA masks can be used to turn bits on. For example, if you wanted to "shift" an ASCII character by setting the seventh bit, you could LDA CHARACTER:ORA #$80. The number $80 in binary is 10000000, so all the bits in CHARACTER which are ORed with zeros here will be left unchanged. (If a bit in CHARACTER is a 1, it stays a 1. If it is a zero, it stays 0.) But the 1 in the seventh bit of $80 will cause a 0 in the CHARACTER to turn into a 1. (If CHARACTER already has a 1 in its seventh bit, it will remain a 1.)

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Immediate	ORA #15	$09/9	2
Zero Page	ORA 15	$05/5	2
Zero Page,X	ORA 15,X	$15/21	2
Absolute	ORA 1500	$0D/13	3
Absolute,X	ORA 1500,X	$1D/29	3
Absolute,Y	ORA 1500,Y	$19/25	3
Indirect,X	ORA (15,X)	$01/1	2
Indirect,Y	ORA (15),Y	$11/17	2

Affected flags: N Z

PHA

What it does: Pushes the Accumulator onto the stack.

Major uses: To temporarily *(very temporarily)* save the byte in the Accumulator. If you are within a particular subroutine and you need to save a value for a brief time, you can PHA it. But beware that you must PLA it back into the Accumulator *before any RTS* so that it won't misdirect the computer to the wrong RTS address. All RTS addresses are saved on the stack. Probably a safer way to temporarily save a value (a number) would be to STA TEMP or put it in some other temporary variable that you've set aside to hold things. Also, the values of A, X, and Y need to be temporarily saved, and the programmer will combine TYA and TXA with several PHAs to stuff all three registers onto the stack. But, again, matching PLAs must restore the stack as soon as possible and certainly prior to any RTS.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Implied	PHA	$48/72	1

Affected flags: none of them .

PHP

What it does: Pushes the "processor status" onto the top of the stack. This byte is the Status Register, the byte which holds all the flags: N Z C I D V.

Major uses: To temporarily *(very temporarily)* save the state of the flags. If you need to preserve the all current conditions for a minute (see description of PHA above), you may also want to preserve the Status Register as well. You must, however, restore the Status Register byte and clean up the stack by using a PLP before the next RTS.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Implied	PHP	$08/8	1

Affected flags: none of them.

PLA

What it does: Pulls the top byte off the stack and puts it into the Accumulator.

Major uses: To restore a number which was temporarily stored on top of the stack (with the PHA instruction). It is the opposite action of PHA (see above). Note that PLA does affect the N and Z flags. Each PHA must be matched by a corresponding PLA if the stack is to correctly maintain RTS addresses, which is the main purpose of the stack.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Implied	PLA	$68/104	1

Affected flags: N Z

PLP

What it does: Pulls the top byte off the stack and puts it into the Status Register (where the flags are). PLP is a mnemonic for PuLl Processor status.

Major uses: To restore the condition of the flags after the Status Register has been temporarily stored on top of the stack (with the PHP instruction). It is the opposite action of PHP (see above). PLP, of course, affects *all* the flags. Any PHP must be matched by a corresponding PLP if the stack is to correctly maintain RTS addresses, which is the main purpose of the stack.

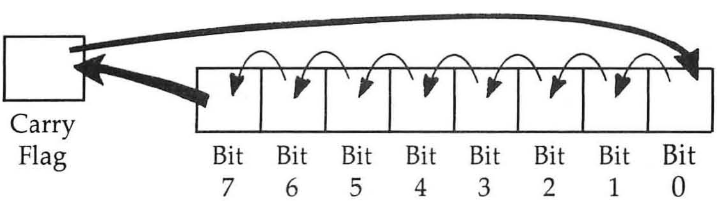
Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Implied	PLP	$28/40	1

Affected flags: all of them.

ROL

What it does: Rotates the bits in the Accumulator or in a byte in memory to the left, by one bit. A rotate left (as opposed to an ASL, Arithmetic Shift Left) moves bit 7 to the carry, *moves the carry into bit 0,* and every other bit moves one position to its left. (ASL operates quite similarly, except it always puts a 0 into bit 0.)



Major uses: To multiply a byte by 2. ROL can be used with ASL to multiply multiple-byte numbers since ROL pulls any carry into bit 0. If an ASL resulted in a carry, it would be thus taken into account in the next higher byte in a multiple-byte number. (See Appendix D.)

Notice how the act of moving columns of binary numbers to the left has the effect of multiplying by 2:

0010 (the number 2 in binary)
0100 (the number 4)

This same effect can be observed with decimal numbers, except the columns represent powers of 10:

0010	(the number 10 in decimal)
0100	(the number 100)

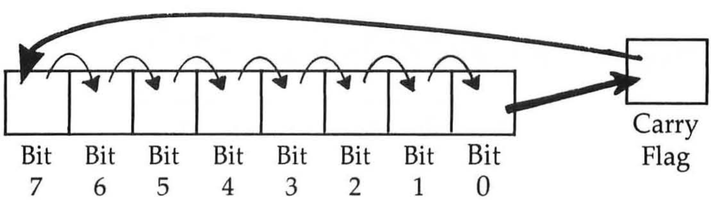
Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Accumulator	ROL	$2A/42	1
Zero Page	ROL 15	$26/38	2
Zero Page,X	ROL 15,X	$36/54	2
Absolute	ROL 1500	$2E/46	3
Absolute,X	ROL 1500,X	$3E/62	3

Affected flags: N Z C

ROR

What it does: Rotates the bits in the Accumulator or in a byte in memory to the right, by one bit. A rotate right (as opposed to a LSR, Logical Shift Right) moves bit 0 into the carry, *moves the carry into bit 7,* and every other bit moves one position to its right. (LSR operates quite similarly, except it always puts a 0 into bit 7.)



Major uses: To divide a byte by 2. ROR can be used with LSR to divide multiple-byte numbers since ROR puts any carry into bit 7. If an LSR resulted in a carry, it would be thus taken into account in the next lower byte in a multiple-byte number. (See Appendix D.)

Notice how the act of moving columns of binary numbers to the right has the effect of dividing by 2:

1000	(the number 8 in binary)
0100	(the number 4)

This same effect can be observed with decimal numbers, except the columns represent powers of 10:

1000	(the number 1000 in decimal)
0100	(the number 100)

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Accumulator	ROR	$6A/106	1
Zero Page	ROR 15	$66/102	2
Zero Page,X	ROR 15,X	$76/118	2
Absolute	ROR 1500	$6E/110	3
Absolute,X	ROR 1500,X	$7E/126	3

Affected flags: N Z C

RTI

What it does: Returns from an interrupt.

Major uses: None. You might want to add your own routines to your machine's normal interrupt routines (see SEI below), but you won't be *generating* actual interrupts of your own. Consequently, you cannot ReTurn from Interrupts you never create.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Implied	RTI	$40/64	1

Affected flags: all of them (Status Register is retrieved from the stack).

RTS

What it does: Returns from a subroutine jump (caused by JSR).

Major uses: Automatically picks off the two top bytes on the stack and places them into the Program Counter. This reverses the actions taken by JSR (which put the Program Counter bytes onto the stack just before leaving for a subroutine). When RTS puts the return bytes into the Program

Counter, the next event in the computer's world will be the instruction following the JSR which stuffed the return address onto the stack in the first place.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Implied	RTS	$60/96	1

Affected flags: none of them.

SBC

What it does: Subtracts a byte in memory *from* the byte in the Accumulator, and "borrows" if necessary. If a "borrow" takes place, the carry flag is cleared (set to 0). Thus, you always SEC (set the carry flag) before an SBC operation so you can tell if you need a "borrow." In other words, when an SBC operation clears the carry flag, it means that the byte in memory was *larger* than the byte in the Accumulator. And since memory is subtracted from the Accumulator in an SBC operation, if memory is the larger number, we must "borrow."

Major uses: Subtracts one number from another.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Immediate	SBC #15	$E9/233	2
Zero Page	SBC 15	$E5/229	2
Zero Page,X	SBC 15,X	$F5/245	2
Absolute	SBC 1500	$ED/237	3
Absolute,X	SBC 1500,X	$FD/253	3
Absolute,Y	SBC 1500,Y	$F9/249	3
Indirect,X	SBC (15,X)	$E1/225	2
Indirect,Y	SBC (15),Y	$F1/241	2

Affected flags: N Z C V

SEC

What it does: Sets the carry (C) flag (in the processor Status Register byte).

Major uses: This instruction is always used before any SBC operation to show if the result of the subtraction was negative (if the Accumulator contained a smaller number than the byte in memory being subtracted from it). See SBC above.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Implied	SEC	$38/56	1

Affected flags: C

SED

What it does: Sets the decimal (D) flag (in the processor Status Register byte).

Major uses: Setting this flag puts the 6502 into decimal arithmetic mode. This mode can be easier to use when you are inputting or outputting decimal numbers (from the user of a program or to the screen). Simple addition and subtraction can be performed in decimal mode, but most programmers ignore this feature since more complicated math requires that you remain in the normal binary state of the 6502.

Note: Commodore computers automatically clear this mode when entering ML via SYS. However, Apple and Atari computers can enter ML in an indeterminant state. Since there is a possibility that the D flag might be set (causing havoc) on entry to an ML routine, it is sometimes suggested that owners of these two computers use the CLD instruction at the start of any ML program they write. Any ML programmer must CLD following any deliberate use of the decimal mode.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Implied	SED	$F8/248	1

Affected flags: D

SEI

What it does: Sets the interrupt disable flag (the I flag) in the processor status byte. When this flag is up, the 6502 will not acknowledge or act upon interrupt attempts (except a few nonmaskable interrupts which can take control in spite of this flag, like a reset of the entire computer). The operating systems of most computers will regularly interrupt the activities of the chip for necessary, high-priority tasks such as updating an internal clock, displaying things on the TV, receiving signals from the keyboard, etc. These interruptions of whatever the chip is doing normally occur 60 times every second. To find out what housekeeping routines your computer interrupts the chip to accomplish, look at the pointer in $FFFE/FFFF. It gives the starting address of the maskable interrupt routines.

Major uses: You can alter a RAM pointer so that it sends these interrupts to *your own ML routine,* and your routine then would conclude by pointing to the normal interrupt routines. In this way, you can add something you want (a click sound for each keystroke? the time of day on the screen?) to the normal actions of your operating system. The advantage of this method over normal SYSing is that your interrupt-driven routine is essentially transparent to whatever else you are doing (in whatever language). Your customization appears to have become part of the computer's ordinary habits.

However, if you try to alter the RAM pointer *while the other interrupts are active,* you will point away from the normal housekeeping routines in ROM, crashing the computer. This is where SEI comes in. You disable the interrupts while you LDA STA LDA STA the new pointer. Then CLI turns the interrupt back on and nothing is disturbed.

Interrupt processing is a whole subcategory of ML programming and has been widely discussed in magazine articles. Look there if you need more detail.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Implied	SEI	$78/120	1

Affected flags: I

STA

What it does: Stores the byte in the Accumulator into memory.

Major uses: Can serve many purposes and is among the most used instructions. Many other instructions leave their results in the Accumulator (ADC/SBC and logical operations like ORA), after which they are stored in memory with STA.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Zero Page	STA 15	$85/133	2
Zero Page,X	STA 15,X	$95/149	2
Absolute	STA 1500	$8D/141	3
Absolute,X	STA 1500,X	$9D/157	3
Absolute,Y	STA 1500,Y	$99/153	3
Indirect,X	STA (15,X)	$81/129	2
Indirect,Y	STA (15),Y	$91/145	2

Affected flags: none of them.

STX

What it does: Stores the byte in the X Register into memory.

Major uses: Copies the byte in X into a byte in memory.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Zero Page	STX 15	$86/134	2
Zero Page,Y	STX 15,Y	$96/150	2
Absolute	STX 1500	$8E/142	3

Affected flags: none of them.

STY

What it does: Stores the byte in the Y Register into memory.

Major uses: Copies the byte in Y into a byte in memory.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Zero Page	STY 15	$84/132	2
Zero Page,X	STY 15,X	$94/148	2
Absolute	STY 1500	$8C/140	3

Affected flags: none of them.

TAX

What it does: Transfers the byte in the Accumulator to the X Register.

Major uses: Sometimes you can copy the byte in the Accumulator into the X Register as a way of briefly storing the byte until it's needed again by the Accumulator. If X is currently unused, TAX is a convenient alternative to PHA (another temporary storage method).

However, since X is often employed as a loop counter, TAX is a relatively rarely used instruction.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Implied	TAX	$AA/170	1

Affected flags: N Z

TAY

What it does: Transfers the byte in the Accumulator to the Y Register.

Major uses: Sometimes you can copy the byte in the Accumulator into the Y Register as a way of briefly storing the byte until it's needed again by the Accumulator. If Y is currently unused, TAY is a convenient alternative to PHA (another temporary storage method).

However, since Y is quite often employed as a loop counter, TAY is a relatively rarely used instruction.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Implied	TAY	$A8/168	1

Affected flags: N Z

TSX

What it does: Transfers the Stack Pointer to the X Register.

Major uses: The Stack Pointer is a byte in the 6502 chip which points to where a new value (number) can be added to the stack. The Stack Pointer would be "raised" by two, for example, when you JSR and the two bytes of the Program Counter are pushed onto the stack. The next available space on the stack thus becomes two higher than it was previously. By contrast, an RTS will pull a two-byte return address off the stack, freeing up some space, and the Stack Pointer would then be "lowered" by two.

The Stack Pointer is always added to $0100 since the stack is located between addresses $0100 and $01FF.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Implied	TSX	$BA/186	1

Affected flags: N Z

TXA

What it does: Transfers the byte in the X Register to the Accumulator.

Major uses: There are times, after X has been used as a counter, when you'll want to compute something using the value of the counter. And you'll therefore need to transfer the byte in X to the Accumulator. For example, if you search the screen for character $75:

```
CHARACTER = $75:SCREEN =
$0400
LDX #0
LOOP LDA SCREEN,X:CMP
#CHARACTER:BEQ MORE:INX
BEQ NOTFOUND                ; (this prevents an endless loop
MORE TXA                    ; (you now know the charac-
                            ter's location)
NOTFOUND BRK
```

In this example, we want to perform some action based on the location of the character. Perhaps we want to remember the location in a variable for later reference. This will require that we transfer the value of X to the Accumulator so it can be added to the SCREEN start address.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Implied	TXA	$8A/138	1

Affected flags: N Z

TXS

What it does: Transfers the byte in X Register into the Stack Pointer.

Major uses: Alters where, in the stack, the current "here's storage space" is pointed to. There are no common uses for this instruction.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Implied	TXS	$9A/154	1

Affected flags: none of them.

TYA

What it does: Transfers the byte in the Y Register to the Accumulator.

Major uses: See TXA.

Addressing Modes:

Name	Format	Opcode	Number of Bytes Used
Implied	TYA	$98/152	1

Affected flags: N Z

Chapter 11

Modifying LADS:

Adding Error Traps, RAM-Based Assembly, and a Disassembler

Modifying LADS: Adding Error Traps, RAM-Based Assembly, and a Disassembler

Special Notes on the Construction of Atari and Apple LADS

Imagine how nice it would be if you could add any additional commands to BASIC that you desired. You wouldn't just temporarily wedge the new commands into a frozen ROM BASIC. Instead, you would simply define the new commands, and they would then become a permanent part of your programming language.

This freedom to change a language is called *extensibility*. It's one of the best features of Forth and a few other languages. Extensibility opens up a language. It gives the programmer easy access to all aspects of his programming tool. LADS, too, is extensible since the internals of the assembler are thoroughly mapped, documented, and explained in this book. You can customize it at will, building in any features that you would find useful.

After exploring the details of the LADS assembler and using LADS to write your own machine language, you may have thought of some features or pseudo-ops that you would like to add. In this chapter, we'll show how to make several different kinds of modifications. These examples, even if they're not features of use to you, will demonstrate how to extend and customize the language. We'll add some new error traps, create a disassembler, and make a fundamental change to the Commodore and Apple LADS—the capability of assembling directly from RAM. (The Atari version has this feature built-in already.)

At the end of this chapter we'll cover the details of the Atari and Apple LADS source code where they differ from the general LADS source listings (printed at the end of each chapter). The three versions—Commodore, Atari, and Apple—are functionally identical, so the descriptions throughout the book apply to each version. However, a few adjustments had to be made: input/output variations, a special source code editor for the Atari, etc. All these will be discussed below. But first, let's see some examples of how to customize LADS.

A Naked Mnemonic Error Trap

The original version of LADS notifies you of most serious errors: branch out of range, duplicated or undefined labels, naked labels (labels without arguments), invalid pseudo-ops, no starting address, file not found on disk, and various syntax errors. Other kinds of errors are forgiven by LADS since it can interpret what you meant to type in your source code. For example, LADS can interpret what you meant when you type errors like these:

```
100 INY #77; (adding an argument to a one-byte opcode)
100 INY     :   LDA #15:INY:INX;(extra spaces before or after
                                 colons)
```

The source code in these examples will be correctly assembled. Also, if you forget to leave a space between a mnemonic and its argument (like: LDA#15), that sort of error will be trapped and announced.

But the original LADS didn't have a built-in trap for naked mnemonics. If you wrote:

```
100 INC:INY:LDA #15         ; (that "INC" requires an argument)
```

the assembler would have crashed. No error message, no warning, just a crash.

Programmers who tested the early versions of LADS asked that this error be trapped. That is, if this mistake was made during the typing of an ML program's source code, it shouldn't cause the assembler to go insane. The following two error-trap modifications have been made a permanent part of LADS (and are already in the object code version you typed in from this book or received on disk).

To expose naked mnemonic errors, a special trap was inserted into the Eval subprogram (see Listing 11.1)

After Eval has determined (line 930 of Program 3-1) that the mnemonic under evaluation *does* require an argument (it's not like INY, which uses Implied addressing and never has an argument), Eval then goes down to check to see if the argument is a label or a number (**1460**).

Here's where we can check to see if the programmer forgot to give an argument. If the mnemonic is followed by a colon or a 0 (end of logical line), that's a sure signal that the argument has been left out. We can load in the character just after the mnemonic (see line 1474, Listing 11.1). If there is a space character (#32), all is well and we can continue (**1480**) with our assembly. If not, we jump to L700, the error-reporting routine which will print the error and ring the bell.

A Trap for Impossible Instructions

Another programmer who tested LADS was just starting to learn machine language. Unfamiliar with some of the mnemonics and addressing modes, he once tried to assemble a line like this:

100 LDA 15,Y

not knowing that Zero Page,Y addressing is a rare addressing mode, exclusively reserved for only two mnemonics: LDX and STX. But LADS didn't crash on this. Instead, it assembled an LDA 15,X (the correct addressing mode, but fatal to his particular program since he was trying to use the Y Register as an index).

The trap was inserted into LADS (Listing 11.2) to make a harmless substitution, to assemble an Absolute,Y (at a zero page address). Thus, the programmer's intent is preserved, but the illegal addressing mode is replaced.

By the time Eval reaches this point, it has already filtered out many other possible addressing modes. Eval knows that the addressing mode is some form of ,X or ,Y and that it's Zero Page. Eval first checks to see if we are dealing with an attempted ,Y addressing mode (CMP #89, the Y character). If not, we continue with the assembly (**5271**) by a BNE to line 5274.

But if it is a ,Y, we check the opcode to see if it is LDX, the only correct opcode for this addressing mode. If so, we continue.

However, if it is some other mnemonic like LDA or STY, this ,Y addressing mode is illegal and we make the adjustment to Absolute,Y by a JMP to the area of Eval where that addressing mode is accomplished.

Most illegal addressing will be reported by LADS. Nevertheless, if there's a peculiar error that you often make when programming and LADS doesn't alert you, just add an error-reporting trap or have the assembler automatically correct the problem.

A final minor modification to the PDISK routine in the Pseudo subprogram will permit embedded keywords in filenames when using the .D pseudo-op to save object code to disk. (The Atari version will not need this modification.) As printed in this book, LADS will correctly extend and print a filename following the .D pseudo-op which contains a keyword. For example, .D OLDSTOP will look correct onscreen. However, LADS will send the tokenized keyword to the disk as the filename. This will result in unpredictable filenames when you use BASIC commands as part of a filename. To correct this, remove line 1190 of Program 8-1 and adjust the following lines in the Pseudo subprogram. Then re-assemble a new version of LADS:

```
1230 PD1 LDY #0
1231 PDLO LDA LABEL,Y:BEQ PDEN:STA FILEN,Y:INY:JMP
     PDLO; MOVE NAME
1239 PDEN LDA #44; PUT ,P,W (PROGRAM, WRITE) SIGNAL
     S ONTO FILENAME
```

Listing 11.1

```
1472 ;------------------- SEE CHAPTER 11 FOR DESCRIPTION OF THIS ERROR TRAP
1473 ;                                 (TRAP FOR NAKED MNEMONICS ERROR)
1474 LDA LABEL+3:CMP #32:BEQ GVEG:JMP L700; (TEST FOR "INC:" TYPE ERROR)
```

Listing 11.2

```
5270 ; ---------- SEE CHAPTER 11 FOR EXPLANATION OF THIS ERROR TRAP ------
5271 L760 LDA BUFFER+2,Y:CMP #89:BNE ML760; --- ERROR TRAP FOR LDA (15,Y)
5272 LDA OP:CMP #182:BEQ ML760; IS THE MNEMONIC LDX (IF SO, MODE IS CORRECT)
5273 JMP L680; IF NOT, JUMP TO MAKE IT (LDA $0015,Y) ABSOLUTE Y
5274 ML760 JMP TWOS
```

A Remarkably Simple, Yet Radical, Change

Since LADS uses symbols instead of numbers, it's fairly easy to change, to make it what you want it to be. What's more, all the programs you write with LADS will also be symbolic and easily changed. Let's make a radical change to LADS and see how easy it is to profoundly alter the nature of the assembler.

As designed, LADS reads source code off a disk program file. Let's make it read its source code from within the computer's RAM memory, instead of from disk. This makes two things possible: 1. You can change source code, then test it by a simple SYS to LADS. 2. Tape drive users can use LADS. This version of LADS isn't functionally different from the normal version since long, linked assembly will still be coming from disk files. However, it can be a more convenient way to write and debug smaller ML programs or subroutines. Everything works the same when you assemble, except that the first (or only) source code program resides in RAM instead of on disk. Commodore and Atari RAM-LADS versions can use linked files, but the Apple RAM-based version cannot link files as it can in the normal Apple LADS.

You make a radical change whenever you change *= 864 to *= 5000. You are making a small change at the beginning, the root, of your source code. After making this change, the entire program is assembled at address 5000 instead of address 864. The effect—in the usual sense of the term—is quite radical. The effort on your part, however, is rather minor. Likewise, we can drastically alter the way that LADS works by making a few minor changes to the symbols in LADS.

Our goal is to make LADS read source code from memory instead of from disk files. First, we need to add two new pointers to the LADS zero page equates (in the Defs file). We create PMEM. It will serve as a dynamic pointer. It will always keep track of our current position in memory as we assemble source code.

The intelligence in the disk drive keeps track of where we are in a file; whenever we call CHARIN, it increments a pointer so that the next CHARIN call will pull a new byte into A, the Accumulator. But we're going to be reading from memory so we'll need to update our own dynamic pointer. To create this pointer, just type in a new line in Defs: PMEM = $xx (whatever zero page, two-byte space is safe in your computer).

The other new pointer we need to define in zero page will tell LADS where your BASIC RAM memory starts, where a program in BASIC starts. To create this register, just look at a map of the zero page of your particular computer and define: RAMSTART = $xx (whatever it is).

Note: These definitions have already been added to the Commodore versions of the Defs subprogram in this book. If you are creating a RAM-based version of LADS for the Apple, add the following two lines to the Apple Defs file:

```
135 RAMSTART = $67; POINTER TO START OF RAM
                              MEMORY
157 PMEM = $E2
```

The Apple version of the RAM-based LADS requires the same changes to the Eval subprogram as Commodore machines require. However, no changes are needed in the Pseudo or Open1 subprograms. The one difference between Commodore and Apple versions in the Getsa subprogram is that Apple requires #$2A in line 300 instead of the #172.

A New CHARIN

Next, we need to change the CHARIN subroutine itself. As LADS normally runs, it goes to BASIC's get-a-byte subroutine whenever CHARIN is invoked. This won't work for memory-based source code. BASIC RAM cannot, alas, be OPENed as if it were a file. So, since LADS is peppered with references to CHARIN, we can just undefine CHARIN in the Defs subprogram by putting a semicolon in front of it (Listing 11.3).

Similarly, CHKIN is scattered throughout LADS to reopen file #1, the read-code-from-disk file. We're not using file #1 in this version of LADS, so we add a semicolon to its definition too (Listing 11.4).

But throughout LADS there are references to these two subroutines. We need to write a new CHARIN and CHKIN to replace the ones we just obliterated. LADS will then have somewhere to go, something to do, as it comes upon CHARINs or CHKINs throughout the code. We do this by adding to the Getsa subprogram (Listing 11.5).

Listing 11.3

```
260 ;CHARIN = $FFE4; PULLS IN ONE BYTE
```

Listing 11.4

```
240 ;CHKIN = $FFC6; OPENS A CHANNEL FOR READ (FILE# IN X)
```

Listing 11.5

```
340 ; --------------------------
350 ; "NEW CHARIN"  ASSEMBLE SOURCECODE FROM MEMORY RATHER THAN DISK.
360 ;  (IMITATES CHARIN FOR DISK)
370 ;  RETURNS WITH NEXT BYTE FROM MEMORY, IN A
380 ; --------------------------
390 CHARIN INC PMEM:BNE INCP1:INC PMEM+1; REPLACES CONVENTIONAL CHARIN/DISK
400 INCP1 STY Y:LDY #0:LDA (PMEM),Y:PHP:LDY Y:PLP:RTS; SAVE STATUS REGISTER
410 CHKIN RTS; REPLACES DISK ROUTINE IN DEFS
```

Line 410 is just an RTS. It's a placebo. We never want to reopen file #1 (CHKIN's normal job), so whenever LADS tries to do that, we JSR/RTS and nothing happens. Something does have to happen with CHARIN, however. CHARIN's job is to fetch the next byte in the source code and give it to the Accumulator. So this new version of CHARIN (**390–400**) increments PMEM, our new RAM memory pointer, saves Y, loads the byte, saves the Status Register, restores Y, restores the Status Register, and returns. This effectively imitates the actions of the normal disk CHARIN, except it draws upon RAM for source code.

Here you can see one of those rare uses for PHP and PLP. There are times when it's not enough to save the A, Y, and X Registers. This is one of those times. INDISK returns to Eval only when it finds a colon (end of source instruction), a semicolon (end of instruction, start of comment), or a zero (end of BASIC program line, hence end of source instruction). When we get a zero when we LDA, the zero flag will be set. But the LDY instruction will reset the zero flag. So, to preserve the effect of LDA on the zero flag, we PHP to store the flags on the stack. Then, after the LDY, we restore the status of the flags, using PLP before we return to the Indisk file. This way, whatever effect the LDA had on the flags will be intact. Indisk can thus expect to find the zero flag properly set if a particular LDA is pulling in the final 0 which signifies the end of a line in the BASIC RAM source code.

After making these substitutions to LADS, we need to remove the two references to Open1 (the routine which opens a disk file for source code reading) in the Eval subprogram. These references are at lines 350 and 4350. We can simply remove them from assembly by putting a semicolon in front of them (Listing 11.6).

Early in Eval, we have a JSR GETSA. This is the GET-Start-Address-from-disk routine. We want to change this to: JSR MEMSA. GETSA isn't needed. MEMSA will perform the same job, but for memory-based source code instead of disk-based source code. MEMSA is found in the Getsa subprogram (Listing 11.7).

The first thing that MEMSA does is to put the start-of-BASIC-RAM pointer into PMEM (our dynamic pointer). This positions us to the first byte in the source code. Then it pulls off enough bytes to point to the * in the start address definition in the source code. This is just what Getsa does for a disk file. The rest of MEMSA is identical to Getsa.

Second Generation LADS

That's it. These few substitutions and LADS will read a source file from RAM memory. You can still use .D NAME to create a disk object code file. You can still send the object code disassembly to a printer with .P. All the other pseudo-ops still work fine. A radical change in ten minutes.

The Getsa subprogram contains a complete, step-by-step description of this disk-to-RAM modification of LADS. After you've made the changes to the source code (and saved them to disk), just load in the normal disk version of LADS, enter Defs as the starting file for assembly, and SYS to LADS. It will grind out a brand new, RAM-based assembler for you.

As always, when making a new version of your LADS assembler, be sure to direct object code to the disk (use the .D pseudo-op) so that you won't overwrite the working LADS in the computer. Also be sure you've given the new version a filename that doesn't already exist on the disk.

Listing 11.6

```
350 ;JSR OPEN1; OPEN READ FILE (SOURCE CODE FILE ON DISK)
4350 ;JSR OPEN1; OPEN INPUT FILE (POINT IT TO THE 1ST BYTE IN THE FILE)
```

Listing 11.7

```
220 ; "MEMSA" GET STARTING ADDRESS FROM MEMORY.  LEAVES DISK POINTING AT-
230 ;                  *= THIS SPACE (START ADDRESS)
240 ; !! INITIALIZES PMEM TO START OF MEMORY
250 ; REPLACES "GETSA" SOURCE CODE FILE TO CREATE RAM-BASED ASSEMBLER.
260 ; -------------------------
270 MEMSA LDA RAMSTART:STA PMEM:LDA RAMSTART+1:STA PMEM+1
280 LDX #3:MEM1 JSR CHARIN:DEX:BNE MEM1; ADD 4 TO PMEM TO POINT TO *=
300 JSR CHARIN:CMP #172:BEQ MMSA
310 LDA #<MNOSTART:STA TEMP:LDA #>MNOSTART:STA TEMP+1:JSR PRNTMESS
320 JMP FIN; GO BACK TO BASIC VIA ROUTINE WITHIN EVAL
330 MMSA RTS
```

A Disassembler

In a perfectly symmetrical universe, with a right hand for every left, and a north pole for every south, you could transform an assembler into a disassembler by just making it run backwards.

Unfortunately, ours is not such a universe. Since LADS turns source code into object code, it would seem possible to tinker with it and adjust it a bit and make it turn object code back into source code, to *disassemble*. Not so. We have to link two new files onto LADS to add a disassembler function: Dis and Dtables.

Personal Programming Style

Dis is an example of how a fairly complex ML program can be constructed using LADS. The relatively few comments reflect my personal style of programming. I find many of the variable names are meaningful enough to make the code understandable, especially since the purpose of the lookup tables in Dtables is fairly easy to see.

The relatively few comments in the compressed code in Dis also allow you to look at more instructions at the same time on the screen. This can help during debugging since you might be able to more quickly locate a fault in the overall logic of a program. Nevertheless, many programmers find such dense code hard to read, hard to debug, and generally inefficient.

Obviously, you should write the kind of source code that works for you. The degree of compression is a matter of programming style and personal preference. Some programming teachers insist on heavy commenting and airy, decompressed coding. Perhaps this emphasis is appropriate for students who are just starting out with computing for the same reasons that penmanship is stressed when students are just starting to learn how to write. But you needn't feel that there is only one programming style. There are many paths, many styles.

How to Use the Disassembler

For convenience, Dis is set to start at 17000. That's an easy number to remember when you want to SYS, CALL, or USR to see a disassembly. The version at the end of this chapter is fully functional, but you might want to make modifications. As printed, it will ask for the start address location in RAM of the object code you want to see listed. Notice that the object code must be residing in RAM to be disassembled. (It would be simple, though, to make a disassembler which operated on disk or tape code.) Then it will disassemble until you hit the STOP or BREAK key. You might want to adjust it— you could have it assemble 20 instructions and then halt until a key was pressed. Or you might want to make it print disassemblies to the printer. Or it could ask for both starting and ending addresses before it begins. To have the disassembler you prefer, just modify the code.

The disassembler is included in this book because it demonstrates compressed LADS source code and it also shows how LADS itself can be expanded while borrowing from existing LADS subroutines like STOPKEY and PRNTNUM.

The source code in other parts of the book is somewhat artificial: Each line contains only one mnemonic followed by a description, a comment about the purpose of that line. Normally, such extensive commentary will not be necessary, and many lines can contain multiple statements separated by colons. Dis is an example of LADS source code as many programmers will probably write it.

To add the disassembler to LADS, change the .END DEFS at the end of the Tables subprogram in LADS to .FILE DIS. This will cause the file for Dis to be assembled along with LADS. Dis will link to Dtables, which ends with .END DEFS to permit the second pass through the combined LADS/Dis code.

Keyboard Input

Let's briefly outline the structure and functions of the disassembler. It starts off by printing its prompt message called DISMESS (**30**). The actual message is located in line 710. PRNTMESS is a subroutine within LADS which prints any message pointed to by the variable TEMP.

Then $3F, the ? symbol, is printed and STARTDIS (**50**) sets the hexflag up so that number printouts will be in hexadecimal. If you prefer decimal, LDA #0 and store it in HXFLAG.

Now there's an input loop to let the user input a decimal start address, character by character. If a carriage return is detected (**90**), we leave the loop to process the number. The number's characters are stored in the LABEL buffer and are also printed to the screen as they are entered (**100**).

When we finish getting the input, the LADS Valdec routine changes the ASCII numbers into a two-byte integer in the variable RESULT. We pick up the two-byte number and store it in the variable SA which will be printed to the screen as the address of each disassembled mnemonic.

Line 150 is a bit obscure. It wasn't originally written this way, but testing revealed that the JSR GB in line 190 would increment the start address right off the bat (before anything was disassembled or printed). At the same time, putting that increment lower in the main loop was inconvenient. So the easiest thing was to simply accept a start address from the user, then decrement it. The disassembler will start off with a start address that is one lower than the user intends, but that early increment will fix things up. Thus, the variable PMEM will hold a number which is one lower than the variable SA. Both these variables are keeping track of where in memory we are currently disassembling. But we've got to distinguish in this way between SA which prints to the screen and PMEM which tells the computer the current location.

Battling Insects

This is a good place to observe that programming is never a smooth trip from the original concept to the final product. No programmer is so well-prepared or knowledgeable that he or she simply sits down and calmly creates a workable program. If you find yourself scratching your head, circling around a bug and not trapping it, spending hours or days trying to see what could possibly be wrong—you're in good company. I've worked with some very experienced, very talented people and have yet to see someone fashion a program without snags. And the more significant and sophisticated the program, the more snags it has.

All that can be done, when you hit a snag, is to single-step through the offending area of your program, or set BRK traps, or puzzle over the source code, or try making some tentative reassemblies (not knowing for sure if your changes will have any salutary effect), or sometimes even toss out an entire subroutine and start over. For example, I wrote the rough draft, the first draft of this disassembler, in about two hours. I didn't have the final version working until I'd spent two full days battling bugs. Some were easy to fix, some were monsters. It took about ten minutes to cure that problem with the start address being one too high. But it took hours to locate an error in the disassembler tables, Dtables.

After the user has input the start address, TEMP is made to point to the LABEL buffer and VALDEC is invoked. VALDEC leaves the result of an ASCII-to-integer conversion in the RESULT variable. That number is stored in PMEM and SA (**140–150**). One final adjustment restores SA to the original number input by the user. SA will only print addresses onscreen; PMEM is the real pointer to the current address during disassembly. The decrementing of PMEM, made necessary by that JSR GB early in the main loop, is not necessary for SA. (SA is not incremented by the GB subroutine.)

GETBYTE: The Main Loop

Now we arrive at the main loop. GETBYTE (**190**) first tests to see if the user wants to stop disassembly via the STOPKEY subroutine (in the Eval subprogram within LADS). Then the GB subroutine (**690**) raises the memory pointer PMEM and fetches a byte from memory. This byte is saved in the FILEN buffer and will act as an index, a pointer to the various tables in the Dtables subprogram. For purposes of illustration, let's assume that the byte we picked up held the number 1. One is the opcode for ORA (Indirect,X). We can trace through the main loop of Dis and see what happens when Dis picks up a 1.

The 1 is transferred to the Y Register (**200**), and we then load whatever value is in MTABLE+1 since we LDA MTABLE,Y and Y holds a 1. This turns out to be the number 2, signifying that we've come upon the second opcode (if the opcodes are arranged in ascending order). Notice that BNE will make us skip over the next couple of lines. Anytime we pull a 0 out of MTABLE it means that there is no valid opcode for that number, and we just print the address, the number, and a question mark ($3F). Then we raise the printout address pointer with INCSA and return to fetch the next byte (**210–220**).

However, in our example, we did find something other than a 0 in MTABLE. We've got a valid opcode. Now we have to find out its addressing mode and print a one- or two-byte argument, depending on that addressing mode. Is it Immediate addressing like LDA #15 (one-byte argument) or Absolute addressing like LDA 1500 (two-byte argument)?

Having found a valid opcode, we now extract the mnemonic from WORDTABLE and print it out (**240–330**). First we multiply our number from MTABLE by 3 since each mnemonic has three letters. The number we found in MTABLE was a 2, so we have a 6 after the multiplication. That means that our mnemonic will start in the sixth position within WORDTABLE. We add 6 to the address of WORDTABLE (**280–290**) and leave the variable PARRAY pointing at the first letter O in WORDTABLE.

Now the SA (current disassembly address) is printed onscreen with PRNTSA and a space is printed (**300**). We then print ORA onscreen, one letter at a time (**310–330**), and print another space. Now we're ready to figure out the addressing mode.

Addressing Type

We had previously saved our original byte (the number 1 in our example) in FILEN (**190**). We now retrieve it, pull out the position value from MTABLE (getting the number 2), and load in the addressing mode type from TYPETABLE (see lines 360–410 in the Dtables subroutine listing at the end of this chapter). It turns out that the number 2 we're using in our example will pull out a number 4 from TYPETABLE. The number 4 identifies this as an Indirect X addressing mode.

Between lines 380 and 410 we have a simple decision structure, much like BASIC's ON-GOTO structure. In our example, the CMP #4 in line 390 will now send us to a routine called DINDX which handles Indirect X addressing.

DINDX (**460**) takes advantage of several routines which print symbols to the screen for us: LEPAR prints a left parenthesis; DOONE fetches and prints the next number in RAM memory (the argument for the current mnemonic); COMX prints a comma and an X; and RIPAR finishes things off with a right parenthesis. Now we have something like this onscreen:

0360 ORA (12,X)

so our disassembly of this particular instruction is complete. We JMP to ALLDONE (**600**) and print a carriage return and start the main loop over again to disassemble the next mnemonic.

Other mnemonics and other addressing modes follow a similar path through Dis as they are looked up in Dtables and then printed out.

By the way, if you look at lines 650-680 on page 296, you'll see a peculiar #" pseudo-op. It allows you to specify a character instead of a number for immediate addressing. In line 650 we need to print a comma to the screen. You could LDA #44 (the ASCII code for a comma) and JSR PRINT.

But if you don't want to look up the ASCII code, LADS will do it for you. Just use a quote after the # symbol: LDA #", (followed by the character you're after; in this case, the comma). The correct value for the character will be inserted into your object code. You can see that we used this pseudo-op to load the value for X, Y,), and (symbols as well, in lines 650-680.

Program 11-1. Dis—The Disassembler

```
10 ; DIS -- DISASSEMBLER
20 *= 17000
30 LDA #<DISMESS:STA TEMP:LDA #>DISMESS:STA TEMP+1:JSR PRNTMESS
40 JSR PRNTCR:LDA #$3F:JSR PRINT
50 STARTDIS LDA #1:STA HXFLAG:LDY #0:STY Y
60 DTM0 JSR CHARIN;                  -- GET START ADDRESS (DECIMAL) --
70 BEQ DTM0
80 CMP #$0D; CARRIAGE RETURN
90 BEQ DMO
100 LDY Y:STA LABEL,Y:JSR PRINT
110 INY:STY Y:JMP DTM0
120 DMO LDA #0:STA LABEL,Y:JSR PRNTCR
130 LDA #<LABEL:STA TEMP:LDA #>LABEL:STA TEMP+1:JSR VALDEC
140 LDA RESULT:STA SA:LDA RESULT+1:STA SA+1
150 LDA RESULT:BNE EF:DEC RESULT+1:BF DEC RESULT; LOWER BY ONE
160 LDA RESULT:STA PMEM:LDA RESULT+1:STA PMEM+1
170 ;
180 ;-------------- PULL IN A BYTE AND SEE IF IT IS A VALID OPCODE
190 GETBYTE JSR STOPKEY:JSR GB:STA FILEN;(SAVE AS INDEX)
200 TAY:LDA MTABLE,Y:BNE DMORE:JSR PRNTSA:JSR PRNTSPACE
210 LDX FILEN:LDA #0:JSR PRNTNUM:JSR PRNTSPACE
220 LDA #$3F:JSR PRINT:JSR INCSA:JMP ALLDONE; NOT A VALID OPCODE
230 ; CONTINUE ON, FOUND A VALID OPCODE-----
240 DMORE STA WORK:LDY #0:STY PARRAY+1:ASL:STA PARRAY:ROL PARRAY+1
250 ;              MULTIPLY Y BY THREE
260 LDA WORK:CLC:ADC PARRAY:STA PARRAY:LDA #0:ADC PARRAY+1:STA PARRAY+1
270 ;              ADD THIS TO WORDTABLE
280 CLC:LDA #<WORDTABLE:ADC PARRAY:STA PARRAY
290 LDA #>WORDTABLE:ADC PARRAY+1:STA PARRAY+1
```

```
300 JSR PRNTSA:JSR PRNTSPACE
310 LDY #0:LDA (PARRAY),Y:JSR PRINT:INY
320 LDA (PARRAY),Y:JSR PRINT:INY
330 LDA (PARRAY),Y:JSR PRINT:JSR PRNTSPACE
340 LDY FILEN:LDA MTABLE,Y; 0 MEANS NO ARGUMENT(INDIRECT OR ACCUMULATOR MODES)
350 TAY:DEY:LDA TYPETABLE,Y:BNE BRANCHES
360 JSR INCSA:JMP ALLDONE
370 BRANCHES LDA TYPETABLE,Y
380 CMP #1:BEQ DIMMED
390 CMP #2:BEQ DABSOL:CMP #3:BEQ DZERO:CMP #4:BEQ DINDX:CMP #5:BEQ DINDY
400 CMP #6:BEQ DZEROX:CMP #7:BEQ DABSOLX:CMP #8:BEQ DABSOLY:CMP #9:BEQ DREL
410 CMP #10:BEQ JDJUMPIND
420 JSR DOONE:JSR COMX:JMP ALLDONE; FALL-THROUGH TO TYPE 11 (ZERO,X)
430 DIMMED LDA #"#:JSR PRINT:JSR DOONE:JMP ALLDONE; IMMEDIATE (TYPE 1)
440 DABSOL JSR DOTWO:JMP ALLDONE:JDJUMPIND JMP DJUMPIND;ABSOLUTE (TYPE 2)
450 DZERO JSR DOONE:JMP ALLDONE; ZERO PG  (TYPE 3)
460 DINDX JSR LEPAR:JSR DOONE:JSR COMX:JSR RIPAR:JMP ALLDONE; IND.X (TYPE 4)
470 DINDY JSR LEPAR:JSR DOONE:JSR RIPAR:JSR COMY:JMP ALLDONE; IND. Y (TYPE 5)
480 DZEROX JSR DOONE:JSR COMX:JMP ALLDONE; ZERO X (TYPE 6)
490 DABSOLX JSR DOTWO:JSR COMX:JMP ALLDONE; ABSOLUTE X (TYPE 7)
500 DABSOLY JSR DOTWO:JSR COMY:JMP ALLDONE; ABSOLUTE Y (TYPE 8)
510 DREL JSR GB:BPL RELPL; RELATIVE (TYPE 8)
520 STA WORK:LDA #$FE:SEC:SBC WORK:STA WORK+1
530 SEC:LDA SA:SBC WORK+1:STA WORK
540 LDA SA+1:SBC #$00:TAX:JSR PRNTNUM
550 LDX WORK:JSR PRNTNUM:JSR INCSA:JSR INCSA:JMP ALLDONE
560 RELPL CLC:ADC SA:ADC #2:STA WORK:LDA #0:ADC SA+1
570 TAX:JSR PRNTNUM
580 LDX WORK:JSR PRNTNUM:JSR INCSA:JSR INCSA:JMP ALLDONE
590 DJUMPIND JSR LEPAR:JSR DOTWO:JSR RIPAR:JMP ALLDONE; IND. JUMP (TYPE 10)
600 ALLDONE JSR PRNTCR:LDX BABFLAG:CPX #1:BCC ALLD1:PLA:PLA:JMP FIN
```

```
610 ALLDl JMP GETBYTE
620 DOONE JSR GB:TAX:LDA #Ø:JSR PRNTNUM:JSR INCSA:JSR INCSA:RTS
630 DOTWO JSR GB:PHA:JSR GB:TAX:LDA #Ø
640 JSR PRNTNUM:PLA:TAX:JSR PRNTNUM:JSR INCSA:JSR INCSA:JSR INCSA:RTS
650 COMX LDA #",:JSR PRINT:LDA #"X:JSR PRINT:RTS
660 COMY LDA #",:JSR PRINT:LDA #"Y:JSR PRINT:RTS
670 LEPAR LDA #"(:JSR PRINT:RTS
680 RIPAR LDA #"):JSR PRINT:RTS
690 GB INC PMEM:BNE DINCPl:INC PMEM+l;REPLACES CONVENTIONAL CHARIN/DISK
700 DINCPl STY Y:LDY #Ø:LDA (PMEM),Y:PHP:LDY Y:PLP:RTS; SAVE STATUS REGISTER
710 DISMESS .BYTE "DISASSEMBLY START ADDRESS (DECIMAL)":.BYTE Ø
720 .FILE DTABLES
```

Program 11-2. Dtables

```
10 ; "DTABLES"   TABLES FOR DISASSEMBLER
20 ;
30 ;        TABLE OF 256 POSSIBLE VALUES  (SOME ARE VALID ADDRESSING MODES)
40 ;
50 MTABLE .BYTE 1 2 Ø Ø Ø 3 4 Ø 5 6 7 Ø Ø 8 9 Ø
60 .BYTE 1Ø 11 Ø Ø Ø 12 13 Ø 14 15 Ø Ø Ø 16 17 Ø
70 .BYTE 18 19 Ø Ø 2Ø 21 22 Ø 23 24 25 Ø 26 27 28 Ø
80 .BYTE 29 3Ø Ø Ø Ø 31 32 Ø 33 34 Ø Ø Ø 35 36 Ø
90 .BYTE 37 38 Ø Ø Ø 39 4Ø Ø 41 42 43 Ø 44 45 46 Ø
100 .BYTE 47 48 Ø Ø Ø 49 5Ø Ø 51 52 Ø Ø Ø 53 54 Ø
110 .BYTE 55 56 Ø Ø Ø 57 58 Ø 59 6Ø 61 Ø 62 63 64 Ø
120 .BYTE 65 66 Ø Ø Ø 67 68 Ø 69 7Ø Ø Ø Ø 71 72 Ø
130 .BYTE Ø 73 Ø Ø 74 75 76 Ø 77 Ø 78 Ø 79 8Ø 81 Ø
140 .BYTE 82 83 Ø Ø 84 85 86 Ø 87 88 89 Ø Ø 9Ø Ø Ø
150 .BYTE 91 92 93 Ø 94 95 96 Ø 97 98 99 Ø 1ØØ 1Ø1 1Ø2 Ø
```

```
160 .BYTE 103 104 0 0 105 106 107 0 108 109 110 0 111 112 113 0
170 .BYTE 114 115 0 0 116 117 118 0 119 120 121 0 122 123 124 0
180 .BYTE 125 126 0 0 0 127 128 0 129 130 0 0 0 131 132 0
190 .BYTE 133 134 0 0 135 136 137 0 138 139 140 0 141 142 143 0
200 .BYTE 144 145 0 0 0 146 147 0 148 149 0 0 0 150 151 0
210 ;
220 ;           TABLE OF MNEMONICS (TIED TO THE NUMBERS IN TABLE ABOVE)
230 ;
240 WORDTABLE .BYTE "XXXBRKORAORAASLPHPORAASLORAASLBPLORAORAASL
250 .BYTE "CLCORAORAASLJSRANDBITANDROLPLPANDROLBIT
260 .BYTE "ANDROLBMIANDANDROLSECANDANDROLRTIEOR
270 .BYTE "EORLSRPHAEORLSRJMPEORLSRBVCEOREORLSRCLIEOR
280 .BYTE "EORLSRRTSADCADCRORPLAADCRORJMPADCRORBVSADC
290 .BYTE "ADCRORSEIADCADCRORSTASTYSTASTXDEYTXASTYSTA
300 .BYTE "STXBCCSTASTYSTASTXTYASTATXSSTALDYLDALDX
310 .BYTE "LDYLDALDXTAYLDATAXLDYLDALDXBCSLDALDYLDALDX
320 .BYTE "CLVLDATSXLDYLDALDXCPYCMPCPYCMPDECINYCMPDEXCPYCMPDEC
330 .BYTE "BNECMPCMPDECCLDCMPCMPDECCPXSBCCPXSBCINC
340 .BYTE "INXSBCNOPCPXSBCINCBEQSBCSBCINCSEDSBCSBCINC
350 ;
360 ;           TABLE OF MODE TYPES (TIED TO THE NUMBERS IN MTABLE ABOVE)
370 ;
380 ; (TYPE 0 = IMPLIED) (1 = IMMEDIATE) (2 = ABSOLUTE) (3 = ZERO PG.)
390 ; (TYPE 4 = INDIRECT X ) (5 = INDIRECT Y) (6 = ZERO X) (7 = ABSOLUTE X)
400 ; (TYPE 8 = ABSOLUTE Y ) (9 = RELATIVE)
410 ; (TYPE 10 = JMP INDIRECT) (11 = ZERO Y)
420 ;
430 TYPETABLE .BYTE 0 4 3 3 0 1 0 2 2 9
440 .BYTE 5 6 6 0 8 7 7 2 4 3
450 .BYTE 3 3 0 1 0 2 2 2 9 5
```

```
460 .BYTE 6 6 Ø 8 7 7 Ø 4 3 3
470 .BYTE Ø 1 Ø 2 2 2 9 5 6 6
480 .BYTE Ø 8 7 7 Ø 4 3 3 Ø 1 Ø 1Ø
490 .BYTE 2 2 9 5 6 6 Ø 8 7 7 4 3 3
5ØØ .BYTE 3 Ø Ø 2 2 2 9 5 6 6
51Ø .BYTE 11 Ø 8 Ø 7 1 4 1 3 3
52Ø .BYTE 3 Ø 1 Ø 2 2 2 9 5 6
53Ø .BYTE 6 11 Ø 8 Ø 7 7 8 1 4
54Ø .BYTE 3 3 3 Ø 1 Ø 2 2 2 9
55Ø .BYTE 5 6 6 Ø 8 7 7 1 4 3
56Ø .BYTE 3 3 Ø 1 Ø 2 2 2 9 5
57Ø .BYTE 6 6 Ø 8 7 7
58Ø .END DEFS
```

Notes on the Structure of Atari LADS

The Atari and Commodore machines have one thing in common—a 6502 microprocessor. The Atari 6502 runs at 1.79 megahertz, making it somewhat faster than the Commodore machines. However, the non-6502 hardware—input/output, graphics, and sound—is entirely different. Although many Atari enthusiasts argue that it is the most powerful available on any 6502-based microcomputer, the operating system of the Atari does not perform basic tasks like input/output in the same manner as Commodore machines. An understanding of these differences is essential to fully understand the Atari LADS source code.

The common tasks machine language programs need to perform with input/output are: open a file, read a character or block of characters from the file, write a character or block of characters to a file, and close the file. With the Commodore operating system (often called the Kernal), there are separate routines for each task. You approach each task by adjusting the Accumulator, X, and Y Registers as necessary, as well as storing any required information into special memory locations (usually in zero page). See the discussion of OPEN1 in Chapter 5 for details. For example, the Commodore OPEN must know where to find the filename, the length of the filename, parameters like read or write, and the device number.

On the Atari, there is just one entry point—$E456, called CIO, for all these tasks. Instead of separate entry points, CIO checks a memory location for the command, a number representing the action to take, such as OPEN, CLOSE, PUT, or GET. Other memory locations hold the starting address of a filename or buffer, and the length of the filename or buffer. Extra locations hold specialized information. Each block of I/O information is called an IOCB, for Input/Output Control Block. There are eight of these IOCBs, numbered 0 to 7. IOCB 0 is reserved for the screen editor, and 7 is usually reserved for language I/O, such as LPRINT in BASIC, or SAVE in the LADS editor.

Although much of LADS is concerned with internal data base-type manipulations, such as looking up a label or converting a mnemonic, there is also a good amount of Commodore-style input/output. Routines like OPEN, CLRCHN, CHKIN, and PRINT are actual ROM entry points on Commodore computers. To avoid complex changes in the source code, Atari LADS has a special file called Kernal (see program listings below), which transparently supports all these routines, making the conversion between the Atari's I/O system and the Commodore's transparent. Explanations of Commodore I/O given in Chapter 5, then, are valid as well for the Atari LADS system. In other words, when the original Commodore version of LADS was translated to the Atari, the Kernal subprogram was added to mimic the operations of the Commodore operating system I/O. This emulation allows the descriptions of LADS to remain essentially identical for non-Commodore machines.

Atari Memory Layout

Memory maps for Commodore computers are relatively simple. Zero page is used by the system, page 1 for the stack, page 2 for operating system storage, and page 3 for the cassette buffer(s). On the Commodore PET, page 4 (starting at address 1024) on up to location 32768 is free RAM. 32768 is the start of screen memory on the PET, and never moves. On the 64, the screen occupies page 4 up to 2047 ($07FF). Free RAM starts at 2048 ($0800) all the way up to 40959 ($9FFF). BASIC in ROM and the operating system start at 40960 ($A000). Although there is hidden memory beneath the ROMs on both the Atari XL series and the Commodore 64, LADS does not use it.

The Atari memory layout is less fixed. Zero page from locations 0 to 127 completely used by the operating system. An applications program like BASIC can use almost all the memory from 128 to 255. Since Atari LADS operates outside the BASIC environment, it is free to use this zero page memory upwards from location $80.

Unlike the PET and 64, Atari machines have no set amount of memory. Atari 400/800 owners have the option of expanding to 48K, without using bank selection or other tricks. Without DOS, free memory starts at $0700 (page 6 is reserved). With DOS, free RAM starts at about $2000. The screen memory, a little over 1K in length, is stored at the top of memory, and is not fixed, due to memory expansion. Many Atari machine language programs store themselves at the bottom of memory, then use memory above themselves to store text or other information. But because LADS stores its labels *below* itself, the Atari version must be located at the top of memory. Since the top of memory with a cartridge (or with 40K of RAM) is $9FFF, and since Atari LADS is about 7K long, $8000 seems to be a good place. If you have a 48K Atari, you may want to reassemble LADS at $A000. The choice of $8000 does exclude Atari owners with less than 40K, but if you have access to a 40K machine, you could reassemble LADS at 8K below the top of memory.

Let's look at the major differences between the Atari LADS and Commodore LADS source code. We won't get into specifics; for that you can refer to the source code itself. The translation of Atari LADS involved two goals: the creation of a powerful assembly development system without making major changes to most of the Commodore LADS source code. Some subprograms needed no changes, others did. Three new subprograms are required by the Atari version: Kernal, System, and Edit.

Here's how all the subprograms in the Atari LADS are linked:
Defs → Eval → Equate → Array → Open1 → Findmn → Getsa → Valdec → Indisk → Math → Printops → Pseudo → Kernal → System → Edit → Tables

Defs. Here we set the origin to $8000. Since we are simulating Commodore I/O, we have to create some label variables such as FNAMELEN (filename length). These are used by the Kernal routines. Other LADS variables like MEMTOP and PMEM are also given zero page definitions for the sake of speed and for indirect addressing. The BABUF, used for holding comments and holding a line in the editor, is defined as $0500. On Commodore machines it is $0200, the address of the BASIC input buffer.

Eval. The first difference between the Commodore and Atari versions of Eval is that instead of reading the filename off the screen, Atari LADS gets the filename from the command line, passed by the editor. The editor has previously set RAMFLAG to 1 if there is no filename. This is a default to RAM-based assembly (your source code is already in memory and need not be read from disk). If RAMFLAG is 0, LADS must assemble from disk. If the RAMFLAG is nonzero, we skip over putting the filename into FILEN, and jump past the JSR OPEN1 in Eval (since there is nothing to open). At the top of Eval, the left margin is set to zero.

Since LADS has complete control of the Atari, no memory needs to be protected from anything, so the top-of-memory pointer need not be lowered.

In FINI, the RAMFLAG is also checked so that JSR OPEN1 is skipped. In FIN, which FINI falls to after the end of the second pass, we send an extra byte out to the object file, if .D was used.

Equate, Array, and Findmn. There was no need to change any of these modules, since they contain no system-specific coding.

Open1. Many changes have also been made to Open1, although a lot of the source code is similar. FDEV and FSECOND hold the device number and secondary address in Commodore LADS. Here they are used to hold the access type (4 for read, 8 for write) and the auxiliary byte (which is zero here). Open1 checks the RAMFLAG to see whether it should load the file after it's been opened, in case memory assembly has been elected. The actual load is done by using part of the editor's load routine. Because of RAMFLAG, we don't need a separate LOAD1 routine.

If the file can't be opened, we call the editor's error message routine, and then return to the editor. The same error handling is performed for all the OPENs.

OPEN2 writes out the binary file header, made up of two 255's, followed by the starting and ending addresses in low byte/high byte format. The origin (the starting address for the object code) is saved in the variable TA. The object code's ending address is known, and stored in LLSA. LLSA is actually one higher than the ending address, which is why we write an extra zero to the end of the file in Eval. This prevents an ERROR 136 when loading the file from DOS.

OPEN4 just opens a file for write to the printer. The printer's filename is P:, which is given in the .BYTE statement as 80 58.

Getsa. Getsa is very similar to the Commodore version. There is no MEMSA—Getsa initializes PMEM to point to the start of the editor's text buffer (TEXTBAS), even if PMEM is not used. Since CHARIN is smart, checking RAMFLAG to decide whether to assemble from memory or from disk, no more changes need to be made.

Valdec. Valdec would have been unchanged from the Commodore version, since there is no machine-specific code. However, the editor makes use of Valdec to convert ASCII line numbers into integers. The ASCII line number does not end with a zero, though. The first part of Valdec finds the length of the number by checking for a zero. It has been changed in the Atari version to exit on any nonnumeric digit (one with an ASCII value less than 48 or greater than/equal to 58). The change does not affect any other use of Valdec.

Indisk. It is in Indisk where we see many modifications to the Commodore version. Since the editor does not tokenize anything, KEYWORD and KEYWAD are not needed, and references to them in this source code, as well as the KEYWORD and KEYWAD routines themselves, have been deleted. Again, since nothing is tokenized, checks for +, *, <, >, etc., look for the ASCII values instead of the tokenized ones. Since line numbers are stored as a string of digits instead of a two-byte integer, we must call LINENUMBER in the SYSTEM module in order to set LINEN. ENDPRO, instead of looking for three zeros to signify the end of a program, must check the disk status variable for end of file. End of file returns 136 after the last character has been read, and $03 if you try to read past the end of file, so we check for both to be safe. We check the status for file #1 (the input file) directly ($0353), instead of ST, since ST may have been changed by another I/O operation. Nonetheless, large parts of Indisk are unchanged from the Commodore version.

Printops. Because of the Kernal simulator, even though Printops has plenty of Commodore I/O calls, few changes were needed to make Printops work on the Atari.

Pseudo. There are some minor changes here. KEYWORD does not need to be used by .END or .FILE. FILE finds the end of the pseudo-op by looking for a space delimiter. The filename is then copied into FILEN, and the file opened. If the current operation is a RAM-based assembly, Open1 takes care of loading in the next file. PEND, which supports .END, first calls FILE to open the file, then copies SA, which holds the current address, into LLSA for use with OPEN2.

Speaking of OPEN2, some code was deleted from PDISK and instead implemented in OPEN2. There were no more changes after PDISK to the Pseudo module. In Commodore LADS, Pseudo links to Tables, the last module. Here we link to Kernal, inserting Kernal, System, and Edit into the chain.

Kernal. This is the most important module in the Atari translation. It implements all the Commodore I/O functions by simulating CHKIN and CHKOUT, and referencing the appropriate IOCB according to FNUM. The CIO equates are first defined: ICCOM, the command byte; ICBADR, which holds the address of the filename or buffer; ICBLEN, which holds the length of the filename or buffer; ICAUX1 and ICAUX2, which need to be set to zero; and CIO itself, that single entry point for all input/output.

A simple routine is X16, which multiplies the Accumulator times 16 and stores it in the X Register. X will be an offset from the first IOCB. Since each IOCB is 16 bytes long, we can use Indexed addressing to change the appropriate IOCB with a statement like STA ICCOM,X.

OPEN is the basic open-file routine. It uses X16 to get the IOCB offset, then stores the filename pointer and filename length into ICBADR and ICBLEN. The command byte for open ($03) is stored in ICCOM, then CIO is called. CIO's error status, which is returned in the Y Register, is saved in ST.

CHKIN changes the default input IOCB, which is used in CHARIN. CHKOUT changes the default output IOCB, which is checked for in PRINT. CLOSE just stores the close command (12) into ICCOM and jumps to CALLCIO, part of OPEN. CLRCHN sets the default INFILE and OUTFILE, as well as FNUM and ST to zero, which makes CHARIN and PRINT use IOCB #0, opened to the screen editor.

PRINT is expected to print the character currently in the Accumulator. It first changes any 13's it sees, which are Commodore carriage returns, into 155's (Atari carriage returns). Another entry point, OBJPRINT, does not transform 13's. This is called when object bytes need to be sent to disk, where you don't want 13's changing into 155's. Depending on OUTFILE, PRINT will automatically use the appropriate IOCB (0 for screen, 2 for object output, 4 for printer output). We then set the buffer length to zero, which tells CIO to expect to find the character to print in the Accumulator. The print text command is used, then we call CIO and restore the X and Y Registers, which were saved when PRINT was entered. This prevents any interference with LADS.

CHRIN is also a busy routine. It first checks RAMFLAG to see whether it should get a byte from an I/O device or from the editor's text memory. If it gets a byte from memory, it must check to see if it has gone past the last byte. If so, we jump straight to FINI in Eval. Otherwise, CHRIN gets a byte from disk or the keyboard. It uses INFILE to decide which IOCB to use, then sets the buffer length to zero. This way it requests a single byte from CIO. If a 155 is returned, it is changed into a zero, which is what LADS looks for as end of line.

There is no "check for BREAK key" routine in Atari ROM, so STOPKEY checks the BREAK key flag, which is set to zero if the BREAK key is pressed. If BREAK was pressed, we execute TOBASIC, which jumps back to the editor.

CLALL is not used by LADS, but is used by the editor to close all files in case of an error. It works like the Commodore CLALL routine, and restores the default I/O (input from keyboard, output to screen) by jumping to CLRCHN.

System. A few more routines are provided here which are not directly supported by the operating system. OUTNUM prints the ASCII number given to it in the X Register, which holds the low byte of the number to print, and the Accumulator holding the high byte. We then call $D9AA, which converts the integer number in locations $D4 and $D5 into floating point, and then call $D8E6, which converts the floating point into a printable ASCII sequence of digits starting at $0580. The routine at $D8E6 sets bit 7 in the last digit of the ASCII numeral string. We print the string, checking and masking off bit 7. LINENUMBER reads the ASCII line number from source code and converts it to an integer, using VALDEC. The result is saved in LINEN.

Tables. The major changes here are that the error messages must be typed in inverse video. One extra variable is defined: LLSA to hold the ending address.

Program 11-3. Kernal

```
100 ICCOM = $0342
110 ICBADR = $0344
120 ICBLEN = $0348
130 ICAUX1 = $034A
140 ICAUX2 = $034B
150 CCLOSE = 12
160 CIO = $E456
170 X16 ASL
180 ASL
190 ASL
200 ASL
210 TAX
220 RTS
```

```
230 ;Opens a file OPEN #FNUM,FDEV,FSECOND,(F
    NAMEPTR)
240 OPEN LDA FNUM
250 JSR X16
260 LDA FNAMEPTR
270 STA ICBADR,X
280 LDA FNAMEPTR+1
290 STA ICBADR+1,X
300 LDA FNAMELEN
310 STA ICBLEN,X
320 LDA #0
330 STA ICBLEN+1,X
340 LDA FDEV
350 STA ICAUX1,X
360 LDA FSECOND
370 STA ICAUX2,X
380 LDA #$03
390 STA ICCOM,X
400 CALLCIO JSR CIO
410 STY ST
420 RTS
430 CHKIN STX INFILE
440 RTS
450 CHKOUT STX OUTFILE
460 RTS
470 CLRCHN LDX #0
480 STX INFILE
490 STX OUTFILE
500 STX FNUM
501 STX ST
510 RTS
520 CLOSE JSR X16
530 LDA #12
540 STA ICCOM,X
550 JMP CALLCIO
560 PRINT CMP #13
570 BNE OBJPRINT
580 LDA #155
590 OBJPRINT STA KASAVE
600 STY KYSAVE
610 STX KXSAVE
620 LDA OUTFILE
630 JSR X16
640 LDA #0
650 STA ICBLEN,X
660 STA ICBLEN+1,X
670 LDA #11
680 STA ICCOM,X
690 LDA KASAVE
```

```
700 JSR CALLCIO
710 LDY KYSAVE
720 LDX KXSAVE
730 LDA KASAVE
740 RTS
750 ;
760 CHARIN STY KYSAVE
770 STX KXSAVE
780 LDA RAMFLAG
790 BEQ CHRIN;If RAMFLAG=0 (False) then get
    byte from device
800 ;Else get byte from memory
810 LDY #0:LDA (PMEM),Y:PHA
820 INC PMEM:BNE NINCP1:INC PMEM+1
830 NINCP1 CLC:LDA PMEM:SBC TEXEND:STA KTEMP
840 LDA PMEM+1
850 SBC TEXEND+1
860 ORA KTEMP:BCC NOTEOF:BEQ NOTEOF
880 JMP FINI
890 NOTEOF LDA #0:STA ST:STA $0353
900 PLA:JMP CHRXIT
910 CHRIN LDA INFILE
920 JSR X16
930 LDA #0
940 STA ICBLEN,X
950 STA ICBLEN+1,X
960 LDA #7
970 STA ICCOM,X
980 JSR CALLCIO
990 CHRXIT LDY KYSAVE
1000 LDX KXSAVE
1010 CMP #155
1020 BNE ZICR
1030 LDA #0
1040 ZICR RTS
1050 STOPKEY PHA
1060 LDA $11
1070 BEQ TOBASIC
1080 PLA
1090 RTS
1100 TOBASIC JMP EDIT
1140 ;
1150 CLALL LDX #7
1160 CLLOOP STX KTEMP:TXA:JSR CLOSE
1170 LDX KTEMP:DEX:BNE CLLOOP
1180 JMP CLRCHN
1190 KASAVE .BYTE 0
1200 KYSAVE .BYTE 0
1210 KXSAVE .BYTE 0
```

```
1220 KTEMP .BYTE 0
1230 .FILE D:SYSTEM.SRC
```

Program 11-4. System

```
170 OUTNUM STX $D4
180 STA $D5
190 JSR $D9AA
200 JSR $D8E6
230 LDY #0
240 ONUMLOOP STY OYSAVE
250 LDA ($F3),Y
260 PHA
270 AND #$7F
280 JSR PRINT
290 PLA
300 BMI ONUMEXIT
310 LDY OYSAVE
320 INY
330 BNE ONUMLOOP
340 ONUMEXIT RTS
360 OYSAVE .BYTE 0
390 LINENUMBER LDY #0
400 LINELOOP JSR CHARIN
410 CMP #32
420 BEQ OUTLINE
430 STA BABUF,Y
440 INY
450 JMP LINELOOP
460 OUTLINE LDA #0
470 STA BABUF,Y
480 LDA #<BABUF
490 STA TEMP
500 LDA #>BABUF
510 STA TEMP+1
520 JSR VALDEC
530 LDA RESULT
540 STA LINEN
550 LDA RESULT+1
560 STA LINEN+1
570 LDY #0
580 RTS
590 .FILE D:EDIT.SRC
```

The Atari LADS Editor

The Atari editor is a whole minilanguage system itself. The source code for this subprogram is well commented and should be understandable as it stands. Since it is not a part of

LADS proper, we'll limit ourselves here to an overview of the major routines.

UMOVE and DMOVE are high-speed memory move routines used to adjust the source code when lines are deleted, added, and so forth. UMOVE can move one range of memory to another, provided that the block to be moved is higher in memory. The range of bytes can overlap so UMOVE can be used as a delete routine. DMOVE moves memory downward, and is used for inserting. If the memory ranges do not overlap, either one can be used. FROML and FROMH hold the start of the block to be moved. DESTL and DESTH are where the block is moved to. LLEN and HLEN are set to hold the length of the block to be moved. These routines use self-modifying code for speed.

EDIT is the entry point for LADS when it is first run, as well as the return point from the LADS assembler. It cleans up the stack, resets the left margin to 2, then stores the addresses of all the editor commands into COMVECT, which is a lookup table used by COMMAND. The BRK interrupt is initialized to point to a special breakpoint entry to the editor. We then check to see if this is the first time EDIT has been entered. If so, we need to NEW out any garbage in memory. The NEW routine sets the end-of-text pointer to point to the beginning of text. No memory is actually cleared.

PROMPT is the entry point for a new line. It prints "LADS Ready", then falls through to ENTER, which is the entry point for a new line without printing a prompt. CHARIN from Kernal gets a byte, which is then processed to remove lowercase, etc. The line is stored in the BABUF, starting at $0500. When a carriage return is detected, an Atari carriage return is added to the end of the line in BABUF, and the length of the line is saved in INLEN. If the length is zero, we go back for another line. The first character of the line is checked. If it is a numeric digit, there must be a line number. If there is no line number, then the line must be a command.

If it is a line number, we call GETLNUM to get the integer value of the line number. GETLNUM also calls FINDLINE to see if that line already exists. If it does, the line is deleted. Then we check to see if there is anything else besides just a line number. If not, we don't insert the line into the source code. Since the line was already deleted, this has the desired effect. We then go back for another line.

COMMAND searches through a table of commands, matching the line the user typed in against the table. If the command is not found, a syntax error message is displayed, and we return to PROMPT. If the command is found, we save the position of whatever's after the command (the argument) in ARGPOS. The command number (COMNUM) is used as an index into COMVECT, which holds the addresses of all commands. We get the address, subtract one from it, then put it on the stack. A RTS then ends up pulling this address off and jumping to it. It's like ON-GOTO in BASIC.

MLIST lists the entire text buffer, from TEXTBAS to TEXEND. A second entry point in MLIST, INLIST, is called by the LIST routine to list a part of a program. We also check here for the BREAK key. MLIST is used by SAVE to list the program to disk, cassette, or the printer.

DOS is spectacularly simple. It just jumps through the DOS vector, location $0A.

FINDLINE is crucial to the editor. It searches through the source code, trying to match the line number given to it (LNUM) against all the ASCII line numbers in the program. It uses Valdec to convert the ASCII line number into an integer. Because of all the ASCII to integer conversions, FINDLINE can be slow on long programs. It returns with BEGPTR pointing to the beginning of the line found, and ENDPTR pointing to the end of the line. If there is no program in memory, it returns with BEGPTR and ENDPTR pointing to the start of text. If the line is not found, BEGPTR and ENDPTR point to the next line greater than the line number searched for. If there is no such line, they point to the end of text. The size of the line found is also calculated for the benefit of the delete routine.

DELETE calls FINDLINE, then calls UMOVE to move memory from the end of the line on top of the beginning of the line. TEXEND is then changed to reflect a shorter program. Many checks have to be made to prevent a crash under conditions such as no program in memory. INSERT is similar to DELETE. It calls DMOVE to insert a gap at the position the line was found.

ERRPRINT is used to display an error message. To be safe, it also closes all files. GETNUM gets and converts an ASCII line number to an integer, using the system ASCII-to-floating-point and floating-point-to-integer routines. The routines return a pointer to the end of the number. This pointer is always kept track of so we can check for new command arguments. GETLNUM uses this routine, then calls FINDLINE.

LIST calls GLIST, which is also used by SAVE. GLIST finds out the line number range you want to list. If there is no line number range given, it goes to MLIST to list the entire program. Otherwise, it has to check for just one line given, or a range of lines. It's complicated, but it works.

OPENFILE is used by SAVE, LOAD, and MERGE. It looks at the argument of the command to get the filename, then calls OPEN within Kernal. If there is an error, we jump to PROMPT. SAVE calls OPENFILE with an 8 for output. It then sets the output file and calls GLIST, which sends the listing out to the current output file. After GLIST returns, the file is closed.

MERGE just sets the input file to the device and jumps to PROMPT. PROMPT keeps requesting input and storing lines until it gets an error. It then closes the file and restores default I/O.

Adding Your Own Editor Commands

The LADS command checks to see if there is a filename, then sets the RAMFLAG accordingly and jumps into EVAL. The SYS command calls GETNUM to get the decimal argument, then stores the address right after a JSR, to which it then falls through, creating a self-modifying indirect JSR. If the routine being called ends in a RTS, control will be returned to PROMPT. You can use SYS to add new editor commands. Just check location $D0, which will point to a position with BABUF ($0500) after the SYS number. You can use $D0 to check for extra arguments within BABUF.

LOAD calls OPENFILE to open the load file for read. It has a second entry point (AFTEROPEN) if the file has already opened. For maximum speed, the program is loaded by calling the CIO get-record routine, which loads in the entire file directly at TEXTBAS, the start of text. Beware, though, that no conversions are done on any of the text, and no checks are made for a legal source file. You could even load and list word processing files. AFTEROPEN is called by Open1 if RAM needs to be reloaded during a memory assembly.

The last routine in the editor handles a BRK instruction entry encountered. It prints a message, uses OUTNUM to display the address where the BRK was found, clears the interrupt flag, cleans the stack, then jumps to the Edit entry point. Edit then links to Tables.

Program 11-5. Editor

```
100 ;Line Editor for LADS
110 ;Charles Brannon 1984
0120 ;
0130 PTR = $CB
0140 TEXTBAS = $2000
0150 ;Move routines
0160 ;
0170 JMP EDIT
0180 FROML .BYTE 0
0190 FROMH .BYTE 0
0200 DESTL .BYTE 0
0210 DESTH .BYTE 0
0220 LLEN .BYTE 0
0230 HLEN .BYTE 0
0240 ENDPOS .BYTE 0
0250 INLEN .BYTE 0
0260 LNUM .BYTE 0 0
0270 TEXTPTR .BYTE 0
0280 COMNUM .BYTE 0
0290 TEXEND .BYTE 0 0
0300 LEN .BYTE 0
0310 YSAVE .BYTE 0
0320 BEGPTR .BYTE 0 0
0330 ENDPTR .BYTE 0 0
0340 FOUNDFLAG .BYTE 0
0350 LINESIZE .BYTE 0 0
0360 SAVEND .BYTE 0 0
0370 SAVBEG .BYTE 0 0
0380 ARGPOS .BYTE 0
0390 ZFLAG .BYTE 0
0400 LCFLAG .BYTE 0
0410 FIRSTRUN .BYTE 0
0420 INDEX = $D0
0430 TMP .BYTE 0
0440 ;
0450 UMOVE LDA FROML
0460 STA MOVLOOP+1
0470 LDA FROMH
0480 STA MOVLOOP+2
0490 LDA DESTL
0500 STA MOVLOOP+4
0510 LDA DESTH
0520 STA MOVLOOP+5
```

```
0530 LDX HLEN
0540 BEQ SKIPMOV
0550 MOV1 LDA #0
0560 MOV2 STA ENDPOS
0570 LDY #0
0580 MOVLOOP LDA $FFFF,Y
0590 STA $FFFF,Y
0600 INY
0610 CPY ENDPOS
0620 BNE MOVLOOP
0630 INC MOVLOOP+2
0640 INC MOVLOOP+5
0650 CPX #0
0660 BEQ OUT
0670 DEX
0680 BNE MOV1
0690 SKIPMOV LDA LLEN
0700 BNE MOV2
0710 OUT RTS
0720 ;
0730 DMOVE LDA HLEN
0740 TAX
0750 ORA LLEN
0760 BNE NOTNULL
0770 RTS
0780 NOTNULL CLC
0790 TXA
0800 ADC FROMH
0810 STA DMOVLOOP+2
0820 LDA FROML
0830 STA DMOVLOOP+1
0840 CLC
0850 TXA
0860 ADC DESTH
0870 STA DMOVLOOP+5
0880 LDA DESTL
0890 STA DMOVLOOP+4
0900 INX
0910 LDY LLEN
0920 BNE DMOVLOOP
0930 BEQ SKIPDMOV
0940 DMOV1 LDY #255
0950 DMOVLOOP LDA $FFFF,Y
0960 STA $FFFF,Y
0970 DEY
0980 CPY #255
0990 BNE DMOVLOOP
1000 SKIPDMOV DEC DMOVLOOP+2
```

```
1010 DEC DMOVLOOP+5
1020 DEX
1030 BNE DMOV1
1040 RTS
1050 ;
1060 EDIT LDX #255;Reset stack
1070 TXS
1071 JSR CLALL
1080 LDA #0;Clear RAMFLAG
1090 STA RAMFLAG
1100 LDA #2;Left margin
1110 STA 82
1120 JSR PRNTCR
1130 ;Store addresses of commands
1140 LDA #<LIST
1150 STA COMVECT
1160 LDA #>LIST
1170 STA COMVECT+1
1180 LDA #<DOS
1190 STA COMVECT+2
1200 LDA #>DOS
1210 STA COMVECT+3
1220 LDA #<INIT
1230 STA COMVECT+4
1240 LDA #>INIT
1250 STA COMVECT+5
1260 LDA #<SAVE
1270 STA COMVECT+6
1280 LDA #>SAVE
1290 STA COMVECT+7
1300 LDA #<LOAD
1310 STA COMVECT+8
1320 LDA #>LOAD
1330 STA COMVECT+9
1340 LDA #<MERGE
1350 STA COMVECT+10
1360 LDA #>MERGE
1370 STA COMVECT+11
1380 LDA #<LADS
1390 STA COMVECT+12
1400 LDA #>LADS
1410 STA COMVECT+13
1420 LDA #<SYS
1430 STA COMVECT+14
1440 LDA #>SYS
1450 STA COMVECT+15
1460 ;Set BRK instr. interrupt to breakpoint
       entry
1470 LDA #<BREAK:STA 518:LDA #>BREAK:STA 519
```

```
1480 LDA FIRSTRUN
1490 BEQ DONEW
1500 JMP PROMPT
1510 DONEW LDA #$CB
1520 STA FIRSTRUN
1530 JMP INIT
1540 NEW LDA #<TEXTBAS;Store beginning location at ending pointer
1550 STA TEXEND
1560 LDA #>TEXTBAS
1570 STA TEXEND+1
1580 JSR CLRCHN;Keyboard/Screen
1590 RTS
1600 INIT JSR NEW
1610 ;
1620 PROMPT LDA #<PMSG;Print prompt
1630 LDY #>PMSG
1640 JSR PRMSG
1650 ENTER LDY #0;Get a line
1660 STY ZFLAG
1670 STY LCFLAG
1680 GETIT JSR CHARIN;a character
1690 LDX ST;Error?
1700 BPL NOERR
1710 CPX #136;End of file?
1720 BEQ EOF;don't print error
1730 CPX #128;same for break key abort
1740 BEQ EOF
1750 JSR ERRPRINT;print other error
1760 EOF JSR CLOSEIT;close down active file
1770 JMP PROMPT;get new line
1780 NOERR CMP #34;A quote toggles the lower case flag
1790 BNE NOTQUOTE
1800 PHA;save quote
1810 LDA LCFLAG;flip lowercase
1820 EOR #1
1830 STA LCFLAG
1840 PLA;restore quote
1850 NOTQUOTE CMP #48;an ASCII "0"?
1860 BNE NOTZ
1870 LDX ZFLAG;if so, check to see if it's a leading zero
1880 BEQ GETIT;if it is, ignore it
1890 NOTZ INC ZFLAG;if we get here, reset leading zero flag
1900 CMP #59;now check for comment
1910 BNE NOTREM
```

```
1920 INC LCFLAG;disable lowercase conversion for rest of line
1930 NOTREM LDX LCFLAG
1940 BNE NOTLOWER;if remflag has been set, don't convert lowercase
1950 AND #127;kill inverse
1960 CMP #97;lowercase "a"
1970 BCC NOTLOWER;if less than, not lowercase
1980 CMP #123;lowercase "z"+1
1990 BCS NOTLOWER;if >=, not lowercase
2000 AND #95;kill bit 5 (127-32=95)
2010 NOTLOWER STA BABUF,Y;store it
2020 INY
2030 CMP #0
2040 BNE GETIT
2050 DEY
2060 LDA #155
2070 STA BABUF,Y
2080 STY INLEN;save length of line
2090 CPY #0
2100 BEQ ENTER;if length=0, blank line, so go back
2110 LDA BABUF;first character: is it a number?
2120 CMP #58
2130 BCS COMMAND;greater than "9", so must be a command
2140 CMP #48;"0"
2150 BCS LINE;greater than "9", but greater than/= "0"?
2160 JMP COMMAND;no, so command
2170 ;Must be a line, so get line number
2180 LINE LDA #255;no offset
2190 JSR GETLNUM
2200 LDA INDEX;INDEX points to first non-numeric digit
2210 STA TEXTPTR;so save it
2220 LDA FOUNDFLAG;if it exists
2230 BNE NODELETE;it not, don't delete it
2240 JSR DELETE
2250 NODELETE LDY TEXTPTR;is there any text on the line?
2260 CPY INLEN;compare to line length
2270 BEQ OVERINS;no text, just delete
2280 JSR INSERT;otherwise insert line
2290 OVERINS JMP ENTER;and get another line
2300 ;
```

```
2310 COMMAND LDA #<COMTABLE;point to start of command table
2320 STA PTR
2330 LDA #>COMTABLE
2340 STA PTR+1
2350 LDY #0;for loop
2360 STY COMNUM;clear command number
2370 LDX #0;for loop
2380 SRCH LDA (PTR),Y;get a character of command table
2390 BEQ COMFOUND;if we get zero here, command is found
2400 CMP #255;or syntax error
2410 BEQ SYNERR
2420 CMP BABUF,X;match with parallel character in line buffer?
2430 BNE NOTFND;if comparison fails, try next command
2440 INX;next character
2450 BACKIN INY
2460 BNE SRCH;bump high byte?
2470 INC PTR+1;yes
2480 JMP SRCH;continue
2490 NOTFND LDA (PTR),Y;if not found, skip past ending zero
2500 BEQ NXTONE
2510 INY
2520 BNE NOTFND
2530 INC PTR+1
2540 JMP NOTFND
2550 NXTONE INC COMNUM;bump up command number
2560 LDX #0;continue search
2570 JMP BACKIN
2580 SYNERR LDA #<SYNMSG;print syntax error
2590 LDY #>SYNMSG
2600 JSR PRMSG
2610 JMP PROMPT
2620 COMFOUND STX ARGPOS
2630 LDA COMNUM;indirect jump to address of command
2640 ASL
2650 TAX
2660 LDA COMVECT,X
2670 SEC
2680 SBC #1
2690 STA TMP
2700 LDA COMVECT+1,X
```

```
271Ø SBC #Ø
272Ø PHA
273Ø LDA TMP
274Ø PHA
275Ø RTS
276Ø ;Command table.  Format:
277Ø ;.BYTE "command" Ø,"command" Ø,255 (255 to end table)
278Ø COMTABLE .BYTE "LIST"
279Ø .BYTE Ø
28ØØ .BYTE "DOS"
281Ø .BYTE Ø
282Ø .BYTE "NEW"
283Ø .BYTE Ø
284Ø .BYTE "SAVE "
285Ø .BYTE Ø
286Ø .BYTE "LOAD "
287Ø .BYTE Ø
288Ø .BYTE "MERGE "
289Ø .BYTE Ø
29ØØ .BYTE "LADS"
291Ø .BYTE Ø
292Ø .BYTE "SYS"
293Ø .BYTE Ø
294Ø .BYTE 255
295Ø ;table will hold address of each command routine in low,high format
296Ø COMVECT .BYTE Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø
297Ø ;
298Ø MLIST LDA #<TEXTBAS;Point to beginning of program
299Ø STA PTR
3ØØØ SEC;get length of program to list
3Ø1Ø LDA TEXEND
3Ø2Ø SBC PTR
3Ø3Ø STA LLEN;into LLEN
3Ø4Ø LDA #>TEXTBAS
3Ø5Ø STA PTR+1
3Ø6Ø LDA TEXEND+1
3Ø7Ø SBC PTR+1
3Ø8Ø STA HLEN;and HLEN
3Ø9Ø INLIST LDA HLEN
31ØØ TAX
311Ø ORA LLEN;both zero?
312Ø BNE DOLIST
313Ø RTS;if so, exit LIST
3131 DOLIST LDA #1:STA 766
314Ø CPX #Ø;high byte zero?
```

```
3150 BEQ LOLST;if so, skip primary pass
3160 LDA #0;for primary pass, list fully
3170 STA LEN
3180 RELIST LDY #0
3190 PRLIST LDA (PTR),Y
3200 JSR PRINT;print a character
3210 LDA ST
3220 BMI OUTLIST;exit on error
3230 INY
3240 CPY LEN
3250 BNE PRLIST
3260 INC PTR+1
3270 DEX;primary pass completed?
3280 BMI OUTLIST;if so, do secondary pass
3290 BNE PRLIST;if not, continue
3300 LOLST LDA LLEN;now list remainder (secondary pass)
3310 STA LEN
3320 JMP RELIST;continue
3330 OUTLIST LDA #0:STA 766:RTS;go back to Ready
3340 ;
3350 DOS JMP (10);DOS Vector
3360 ;
3370 FINDLINE LDA #<TEXTBAS;start at top of program
3380 STA PTR;initialize pointer
3390 LDA #>TEXTBAS;same for high bytes
3400 STA PTR+1
3410 LDA #0
3420 STA FOUNDFLAG;set foundflag to affirmative
3430 TAY
3440 ;
3450 LEQ STY YSAVE;preserve Y
3460 TYA;point to first byte in line
3470 CLC
3480 ADC PTR
3490 STA TEMP;so we can convert line #
3500 STA BEGPTR;save start of line
3510 STA ENDPTR
3520 LDA PTR+1;same for high byte
3530 ADC #0
3540 STA TEMP+1
3550 STA BEGPTR+1
3560 STA ENDPTR+1
3570 ;check to see if at end
3580 SEC
```

```
3590 LDA BEGPTR
3600 SBC TEXEND
3610 STA TMP
3620 LDA BEGPTR+1
3630 SBC TEXEND+1
3640 ORA TMP
3650 BCC NOTEND
3660 JMP NOTFOUND2
3670 NOTEND JSR VALDEC
3680 SEC;see if line number matches
3690 LDA RESULT
3700 SBC LNUM
3710 STA TMP
3720 LDA RESULT+1
3730 SBC LNUM+1
3740 ORA TMP
3750 BEQ FOUNDLINE;if match, line found
3760 BCS NOTFOUND
3770 ;no match at all, so continue search
3780 NEXTLINE JSR EOL;skip to end of line
3790 INY;skip over eol
3800 BNE NOADJ2
3810 INC PTR+1
3820 NOADJ2 JMP LEQ;continue search
3830 FOUNDLINE DEC FOUNDFLAG;set to found (a
     fter INC in NOTFOUND2)
3840 NOTFOUND JSR EOL;skip past end of line
3850 CLC;store at ending address
3860 TYA
3870 ADC PTR
3880 STA ENDPTR
3890 LDA #0
3900 ADC PTR+1
3910 STA ENDPTR+1
3920 NOTFOUND2 INC FOUNDFLAG;if 255, then 0
     (found), else 1 (not found)
3930 SEC;get size of line
3940 LDA ENDPTR
3950 SBC BEGPTR
3960 STA LINESIZE;put it in LINESIZE
3970 LDA ENDPTR+1
3980 SBC BEGPTR+1
3990 STA LINESIZE+1
4000 INC LINESIZE
4010 BNE NOINC3
4020 INC LINESIZE+1
4030 NOINC3 RTS
4040 ;skip past end of line
4050 EOL LDY YSAVE;restore Y
```

```
4060 SRCHEND LDA (PTR),Y;get character
4070 CMP #155
4080 BEQ ENDLINE;if zero (EOL)
4090 INY;bump up pointer
4100 BNE SRCHEND;zero?
4110 INC PTR+1;next block
4120 NOADJ JMP SRCHEND;end of line?
4130 ENDLINE RTS
4140 ;
4150 ;Print message
4160 PRMSG STA PTR;prepare pointer
4170 STY PTR+1
4180 LDY #0
4190 PRLOOP LDA (PTR),Y;get msg char
4200 BEQ OUTMSG;zero (end of message)
4210 JSR PRINT;else print char
4220 INY;continue loop
4230 BNE PRLOOP
4240 OUTMSG RTS
4250 ;
4260 ;FINDLINE has initialized BEGPTR, ENDPTR, and LINESIZE
4270 DELETE LDA ENDPTR;move FROM [end of line+1]
4280 CLC
4290 ADC #1
4300 STA FROML
4310 LDA ENDPTR+1
4320 ADC #0
4330 STA FROMH
4340 LDA BEGPTR;to beginning of line
4350 STA DESTL
4360 LDA BEGPTR+1
4370 STA DESTH
4380 SEC;length of move is TEXEND-ENDPTR
4390 LDA TEXEND
4400 SBC ENDPTR
4410 STA LLEN
4420 LDA TEXEND+1
4430 SBC ENDPTR+1
4440 BCS ZLAST
4450 LDA TEXEND
4460 BEQ NODEC2
4470 DEC TEXEND+1
4480 NODEC2 DEC TEXEND
4490 JMP NOMOV
4500 ZLAST STA HLEN
4510 ORA LLEN
4520 BEQ SKIPDEL;nothing to move!
```

```
4530 JSR UMOVE
4540 NOMOV SEC
4550 LDA TEXEND;subtract size of deleted line from program end
4560 SBC LINESIZE
4570 STA TEXEND
4580 LDA TEXEND+1
4590 SBC LINESIZE+1
4600 STA TEXEND+1
4610 SKIPDEL RTS;delete done!
4620 ;
4630 INSERT LDA BEGPTR;insert gap at found line position
4640 STA PTR;also set pointer
4650 STA FROML;move From BEGPTR
4660 SEC
4670 ADC INLEN;to BEGPTR+INLEN+1
4680 STA DESTL
4690 LDA BEGPTR+1
4700 STA PTR+1;same for high
4710 STA FROMH
4720 ADC #0
4730 STA DESTH
4740 SEC;# of bytes to move is
4750 LDA TEXEND;(TEXEND-BEGPTR)+1
4760 SBC BEGPTR
4770 STA LLEN
4780 LDA TEXEND+1
4790 SBC BEGPTR+1
4800 STA HLEN
4810 BCS NOTLAST
4820 LDA TEXEND
4830 BNE NODEC
4840 DEC TEXEND+1
4850 NODEC DEC TEXEND
4860 JMP INSEXIT
4870 NOTLAST ORA LLEN
4880 BEQ INSEXIT;nothing to insert!
4890 NOINC2 JSR DMOVE;do insert
4900 INSEXIT SEC;add length of line added
4910 LDA TEXEND;to end of text pointer
4920 ADC INLEN
4921 STA TEXEND
4940 LDA TEXEND+1
4950 ADC #0
4960 STA TEXEND+1
4970 LDY #0;gap ready, put in line
4980 INSLOOP LDA BABUF,Y
4990 STA (PTR),Y
```

```
5000 INY
5010 CPY INLEN
5020 BCC INSLOOP
5030 BEQ INSLOOP
5040 RTS;insert done!
5050 CLOSEIT LDA FNUM
5060 BEQ NOCLOSE
5070 JSR CLOSE
5080 NOCLOSE JSR CLRCHN
5090 RTS
5100 ERRPRINT LDA ST
5110 STA TMP
5120 JSR CLALL
5130 LDA #<ERRMSG
5140 LDY #>ERRMSG
5150 JSR PRMSG
5160 LDX TMP
5170 LDA #0
5180 JSR OUTNUM
5190 JSR PRNTCR
5200 RTS
5210 PMSG .BYTE 155
5220 .BYTE "LADS Ready."
5230 .BYTE 155 0
5240 SYNMSG .BYTE 253
5250 .BYTE "Syntax Error"
5260 .BYTE 155 0
5270 ERRMSG .BYTE 253
5280 .BYTE "Error - "
5290 .BYTE 0
5300 BRKMSG .BYTE "BRK from "
5310 .BYTE 0
5320 ;
5330 GETNUM STA $F2
5340 INC $F2
5350 LDA #<BABUF;point to line buffer
5360 STA $F3
5370 LDA #>BABUF
5380 STA $F4;offset should be in $f2
5390 JSR $D800;convert ASCII to floating point
5400 BCS NUMERR
5410 JSR $D9D2;floating point to integer
5420 LDA $F2;store pointer to first non-numeral
5430 STA INDEX
5440 RTS
5450 NUMERR LDA #0;clear result
5460 STA $D4
```

```
5470 STA $D5
5480 RTS
5490 GETLNUM JSR GETNUM;Get number from BABUF+(accumulator+1)
5500 LDA $D4;put it in LNUM
5510 STA LNUM
5520 LDA $D5
5530 STA LNUM+1
5540 JSR FINDLINE;find the line
5550 RTS
5560 LIST JSR GLIST
5570 JMP PROMPT
5580 GLIST LDA ARGPOS;Any arguments?
5590 CMP INLEN;not if argpos is at end of line
5600 BNE YESARG
5610 JMP MLIST;so list all
5620 YESARG JSR GETLNUM;get first numeric argument
5630 LDA BEGPTR;list from beginning of first line
5640 STA SAVBEG;save beginning pointer
5650 LDA BEGPTR+1
5660 STA SAVBEG+1
5670 LDA ENDPTR;save end of first line
5680 STA SAVEND
5690 LDA ENDPTR+1
5700 STA SAVEND+1
5710 LDA INDEX;point to second argument
5720 CMP INLEN;if equal, no second argument
5730 BNE YESARG2
5740 LDA FOUNDFLAG;no second arg, so check for legal line
5750 BNE NOLIST;line wasn't found, so don't list it
5760 LDA SAVEND;restore end of line
5770 STA ENDPTR
5780 LDA SAVEND+1
5790 STA ENDPTR+1
5800 JMP OVER2;and skip
5810 YESARG2 JSR GETLNUM;get second line number
5820 OVER2 LDA SAVBEG
5830 STA PTR
5840 LDA SAVBEG+1
5850 STA PTR+1
5860 SEC;calculate length
5870 LDA ENDPTR
5880 SBC PTR
```

```
5890 STA LLEN
5900 LDA ENDPTR+1
5910 SBC PTR+1
5920 STA HLEN
5930 BCS GOLIST;if second # < first#, don't list
5940 NOLIST RTS
5941 GOLIST LDA FOUNDFLAG:BNE NOINCH
5950 INC LLEN
5960 BNE NOINCH
5970 INC HLEN
5980 NOINCH JMP INLIST
5990 ;
6000 OPENFILE CLC
6010 LDA ARGPOS
6020 ADC #<BABUF
6030 STA FNAMEPTR;point to filename
6040 LDA #0
6050 ADC #>BABUF
6060 STA FNAMEPTR+1
6070 LDY ARGPOS;find end of filename
6080 GETFNAME LDA BABUF,Y
6090 CMP #155;end of line?
6100 BEQ ENDFNAME;if so, exit loop
6110 CMP #44;end of filename?
6120 BEQ ENDFNAME;also legal
6130 INY
6140 BNE GETFNAME;if no delimiter found...
6150 JMP SYNERR;it's a syntax error
6160 ENDFNAME TYA;convert Y pointer to length
6170 SEC
6180 SBC ARGPOS;Y-argpos
6190 STY ARGPOS;reset argpos for list
6200 STA FNAMELEN;filename length
6210 LDA #7;CLOSE #7
6220 STA FNUM
6230 JSR CLOSE
6240 LDA #0;OPEN #7,n,0,filename
6250 STA FSECOND
6260 JSR OPEN;do open
6270 LDX ST;check for error
6280 BMI ERRABORT;yes, error
6290 RTS
6300 ERRABORT PLA;disk error, so abort
6310 PLA
6320 JSR ERRPRINT
6330 JMP PROMPT
6340 SAVE LDA #8;8 means output
```

```
6350 STA FDEV
6360 JSR OPENFILE;open the file
6370 LDX FNUM;all PRINTs go
6380 JSR CHKOUT;to file
6390 JSR GLIST;send out listing
6400 JSR CLOSEIT;close file
6410 JMP PROMPT
6420 MERGE LDA #4;4 for input
6430 STA FDEV
6440 JSR OPENFILE;open it
6450 LDX FNUM;all input comes from this file
6460 JSR CHKIN
6470 JMP ENTER;file will be closed automatic
     ally
6480 LADS LDA ARGPOS;Any argument?
6490 CMP INLEN
6500 BNE NOTMEM;if argpos<>inlen, then there
      is,so don't change RAMFLAG
6510 INC RAMFLAG
6520 NOTMEM JMP START
6530 SYS LDA ARGPOS;locate number
6540 JSR GETNUM;get it
6550 LDA $D4;put address directly
6560 STA JUMPVEC+1;into code
6570 LDA $D5;self-modifying!
6580 STA JUMPVEC+2
6590 JUMPVEC JSR $FFFF;this address will be
     changed by above
6600 JMP PROMPT
6610 LOAD JSR PLOAD;do load
6620 JMP PROMPT;done
6630 PLOAD LDA #4;4 for read
6640 STA FDEV
6650 JSR OPENFILE;open file
6660 AFTEROPEN LDA FNUM;all input comes from
      this file
6670 JSR X16
6680 LDA #<TEXTBAS
6690 STA ICBADR,X
6700 LDA #>TEXTBAS
6710 STA ICBADR+1,X
6720 LDA #0
6730 STA ICBLEN,X
6740 LDA #$50
6750 STA ICBLEN+1,X
6760 LDA #7
6770 STA ICCOM,X
6780 JSR CALLCIO
6790 LDA FNUM
```

```
6800 JSR X16
6810 CLC;add buffer length to get ending addr
6820 LDA ICBLEN,X
6830 ADC #<TEXTBAS
6840 STA TEXEND;update end
6850 LDA ICBLEN+1,X
6860 ADC #>TEXTBAS
6870 STA TEXEND+1
6880 LDA ST
6890 CMP #136;end of file?
6900 BEQ NOPRERR;if so, don't print an error message
6910 JSR ERRPRINT
6920 JMP PROMPT
6930 NOPRERR JSR CLOSEIT;close down file
6940 RTS;end of load
6950 BREAK CLI:LDA #<BRKMSG:LDY #>BRKMSG:JSR PRMSG
6960 PLA:PLA:PLA:SEC:SBC #2:TAX:PLA:SBC #0:JSR OUTNUM
6965 LDX #255:TXS:JSR PRNTCR:JMP EDIT
6970 .FILE D:TABLES.SRC
```

Atari Machine Language Programming

There is a lot to be learned from the Atari LADS source code. Both the assembler and the editor are complex, powerful programs. You might find uses in your own programming for such general-purpose routines as Valdec, UMOVE, and DMOVE. You can add functions to the editor such as search and replace. Or you could simply bypass the editor altogether, creating LADS-compatible source files using an ordinary word processor (and thus have access to the search and replace and other features of the word processor program).

Since maps are invaluable in sophisticated ML programming, you might want to purchase *Mapping the Atari* (COMPUTE! Books, 1983).

Special Apple Notes

The Apple version of LADS works the same as the Commodore 64 version with only slight modifications. The Apple doesn't have the convenience of Kernal routines to access DOS, so routines had to be written which could directly access the DOS file manager routines. This required extensive changes to the Open1 subprogram, which are discussed below.

Also, because the Applesoft tokenize routine takes the spaces out of the text, it was necessary to put a wedge into Apple's CHRGET routine to intercept the BASIC tokenize routine. And the wedge includes a routine that puts the filename of the program you want to assemble to the top of the screen where LADS expects to find it.

Apple Disk Access

The Apple DOS file manager is the part of DOS that handles all file input and output to the disk. It calls RWTS (Read/Write to Track/Sector) and is called from the command interpreter. The command interpreter sends control bytes to the file manager through the file manager parameter list. You can access the file manager directly by sending it the parameters it requires.

To get the address of the parameter field you JSR to $03DC. This loads the Accumulator with the high byte and the Y Register with the low byte of the parameter field. You can then store these to a zero page location for easy transfer of the parameters.

Table 11-1. Apple File Manager Parameter List

	Parameter											
	1	2	3/4	5	6	7	8	9/10	11	13/14	15/16	17/18
OPEN	1	*	*	*	*	*	*	*	*	*		
CLOSE	2								*	*	*	*
DELETE	5			*	*	*	*	*	*	*	*	*
CATALOG	6				*	*			*	*		
LOCK	7			*	*	*	*	*	*	*	*	
UNLOCK	8			*	*	*	*	*	*	*	*	
RENAME	9		*	*	*	*	*	*	*	*	*	
INIT	11	157		*	*	*			*	*		
VERIFY	12			*	*	*	*	*	*	*	*	*

	Parameter									
	1	2	3/4	5/6	7/8	9/10	11	12/14	15/16	17/18
READ 1 Byte	3	1				*	*	*	*	*
READ Range	3	2			*	*	*	*	*	*
POSITION and READ 1 Byte	3	3	*	*		*	*	*	*	*
POSITION and READ Range	3	4	*	*	*	*	*	*	*	*
WRITE 1 Byte	4	1				*	*	*	*	*
WRITE Range	4	2			*	*	*	*	*	*
POSITION and WRITE 1 Byte	4	3	*	*		*	*	*	*	*
POSITION and WRITE Range	4	4	*	*	*	*	*	*	*	*
POSITION	10		*	*			*	*		

Note: The numbers in the leftmost column represent the opcode; the numbers across the top of this chart represent byte positions relative to the start of the parameter list. Asterisks signify that a byte is required for the operation listed. A blank space means that this parameter can be ignored. Nevertheless, the byte positions must be maintained. For example, to DELETE, you do not need to worry about the second, third, or fourth bytes—anything can be in them—but they must exist. The first byte must contain a five, and the fifth through the eighteenth bytes must be set up as described below.

The parameters are expained in sections. The first section tells you about all the opcodes except for the read, write, and positions opcodes, because they are slightly different from the rest. The second section tells you about the read, write, and position opcodes; the third, about the last set of parameters that is common to all opcodes.

The first byte of the parameter field is the opcode type. This parameter can be in the range of 1 to 12.

The second parameter is used only with the INIT opcodes. If you are using a 48K Apple, the correct value for this parameter is 157.

The third and fourth parameters are used with the OPEN and RENAME opcodes. Together they hold the record length of a random access file. If you are not using a random access file, you should have a zero in both of these locations. With the RENAME opcode, these bytes hold the address of the new name.

The fifth byte holds the volume number. The sixth byte holds the drive number. The seventh byte holds the slot number. The eighth byte holds the file type.

The ninth and tenth bytes hold the address of the filename. The filename must be stored in the address pointed to by these bytes. It must be padded with spaces.

This section explains the read, write, and position opcodes.

The first byte holds the opcode. The second byte holds the subcode.

The next four bytes are used only when you require a position command. The third and fourth bytes hold the record number. The fifth and sixth bytes hold the byte offset. To reposition the pointer in an open file, you can use these bytes to calculate a new position. The new position is equal to the length of the file specified in the open opcode times the record number plus the byte offset.

The seventh and eighth bytes hold the length of the range of bytes. This is used only when reading or writing a range.

When reading or writing a range of bytes, the ninth and tenth bytes hold the start address of the range. If you are reading or writing only one byte, then the ninth byte holds the byte you read or the byte you are going to write.

The following are parameters for all the opcodes.

The eleventh byte is the error byte. It should be checked each time after you access the file manager. The errors are as follows:

0: NO ERROR
2: INVALID OPCODE
3: INVALID SUBCODE
4: WRITE PROTECTED
5: END OF DATA
6: FILE NOT FOUND
7: VOLUME MISMATCH
8: I/O ERROR
9: DISK FULL
10: FILE LOCKED

The twelfth byte is unused. The thirteenth and fourteenth bytes are used for the address of the work area buffer. This is a 45-byte buffer in one of the DOS buffers.

The fifteenth and sixteenth bytes hold the address of the track/sector list sector buffer. This is a 256-byte buffer in one of the DOS buffers.

The seventeenth and eighteenth bytes hold the address of the data sector buffer. This is another 256-byte buffer in one of the DOS buffers.

Once you have sent the correct parameters, you can call the file manager by a JSR to $03D6. You must specify if you want to create new file on disk if the one you are accessing doesn't exist. This is done by loading the X Register with a 0. If you don't want to create a new file, you can load the X Register with a 1. If you don't want to create a new file and you try to access a file that doesn't exist, you will receive an error number 6 in byte 11 of the parameter field.

Apple LADS uses the routines in the file manager that read or write one byte from or to the disk at a time. The general routine to transfer the parameters from Tables to the file manager can be found between lines 810 and 920 in the Open1 listing. This is called from the individual subroutines for opening, closing, reading, and writing. The OPEN routines require a filename. Lines 580–800 handle the transfer of the filename from the filename buffer to the specific buffer.

There is also a check to see whether a file about to be opened has been opened previously. This was needed because you cannot close a file unless it was previously opened. This is handled in the close routine (**370–570**).

The PRINT routine handles all output, and the CHARIN routine handles all input. There is one input and one output channel, and all input and output must be handled through a channel. The bytes which govern this event are set in the CHKIN and CHKOUT routine. The CHKIN routine (**930–940**) sets all input to come from that file. The CHKOUT routine (**950–1030**) sets all output to go to that file. The PRINT routine (**1170–1430**) and the CHARIN routine (**1040–1160**) check to see what channel is currently open, then go to that routine.

The BASIC wedge (**1700–2530**) handles the tokenizing of the BASIC text. It checks to see if the text pointer is at $200 (the input buffer). If not, it goes to the normal GETCHR routine. Otherwise, it checks to see if the first character is a number. If so, it goes to the insert line routine, and if not, it checks for the characters ASM. If that is found, the wedge concludes its work by putting the filename at the top of the screen and jumping to the start of LADS.

The insert line routine gets the line number, then jumps to the Apple tokenize routine, which loads the Y Register with the length of the line plus six and then jumps to the normal line insert and tokenize routine.

The last subroutine in Open1 is the first thing that is called when you BRUN LADS. It initializes the wedge and sets HIMEM to the start of LADS.

Appendices

How to Use LADS

Here is a step-by-step explanation of how to assemble machine language programs using the LADS assembler. As you familiarize yourself with its features and practice using it, you will likely discover things about the assembler which you'll want to change or features you'll want to add. For example, if you find yourself frequently using an impossible addressing mode like LDY (15,Y), you might want to insert an error trap for that into LADS source code. Chapter 11, "Modifying LADS," shows you how these customizations can be accomplished. But here is a description of the features which are built into LADS.

Apple and Atari Versions

For the most part, the commands and features of LADS are the same for all versions: Apple, Atari, and Commodore. A few differences are discussed at the end of the general instructions for all versions of LADS. No matter which computer you use, you should read the body of this chapter to understand how to get the most out of LADS. Then, if you use an Atari or an Apple, you can read the special notes at the end of this appendix which explain some minor variations applicable to those computers.

General Instructions for Using LADS

LADS assembles from *source files*. They are particularly easy and convenient to create; just turn on your computer and pretend you're writing a BASIC program. (To create source files for the Atari, see "Special Atari Notes" below.) Commodore and Apple LADS work with source files created exactly the way you would write a BASIC program. Here's an example:

```
10 *= $0360
15 .S
20 LDA #22:LDY #0
30 STA $1500,Y
40 .END TEST
```

Use line numbers, colons, and whatever programmer's aids (Toolkit, BASIC Aid, POWER, automatic line numbering, etc.) you ordinarily use to write BASIC itself.

After you've typed in a program, save to disk in the normal way. (Tape drive users: See special "Note to Tape Users" at the end of this appendix.) Notice line 10 in the example above. The first line of any LADS source file must provide the starting address, the address where you want the ML program to begin in the computer's memory. You signify this with the *= symbol, which means "program counter equals." When LADS sees *=, it sets the Program Counter to the number following the equals sign. Remember that *there must be a space between the = and the starting address.*

The last line of each LADS source file must contain either the .END pseudo-op or the .FILE pseudo-op. Both of them link source files together in case you want to chain several files into one large ML program. However, .FILE names the next linked source file in the chain whereas .END always specifies the first source file of the chain. If there is only one file (as in our example above), you still must end it with .END and give its name as the first file. More about this shortly.

Also notice that you can use either decimal or hexadecimal numbers interchangeably in LADS. Lines 10 and 30 contain hex; line 20 has decimal numbers.

After you've saved the source code to disk, you can assemble it by loading LADS and then typing the name of the source file in the upper left-hand corner of the screen. (The Atari version differs here as well.) Let's go through the process step by step. Type in the little source program above as if you were writing a BASIC program. SAVE it by typing:

	SAVE "TEST",8
Then	**LOAD "LADS",8,1**
Type	**NEW**

Clear the screen and type in the source file's name in the upper left-hand corner:

TEST

Then cursor down a line or two and type SYS 11000 and hit the RETURN key. That will activate LADS on the Commodore 64, VIC-20, and 8032 PET/CBM. See the special notes below for using the Atari and Apple versions of LADS.

You will see the assembler create the *object code*, the bytes which go into memory and comprise the ML program.

Note: Be sure to remember that every source code program must end with the **.END NAME** *pseudo-op. In our example, we* *concluded with* **.END TEST** *because TEST is the name of the only file in this source code. Also notice that you do not use quotes with these filenames.*

To review: Every source code program must contain the starting address in the first line (for example, 10 *= $0800) and must list the filename on the last line (for example, 500 .END SCREENPROG). If you chain several source code programs together using the .FILE pseudo-op, you end *only the final program in the chain* with the .END pseudo-op. These two rules will become clearer in a minute when we discuss the .END and .FILE pseudo-ops.

Features

There are a number of *pseudo-ops* (direct instructions to the assembler) available in LADS. The .S in line 15 is such an instruction. It tells LADS to print the results of an assembly to the screen. If you add the following lines to our test program, you will cause the listing to be in decimal instead of hex and cause LADS to save the object code (the runnable ML program) to a disk file called T.OBJ.

```
10 *= $0360
11 .NH
12 .D T.OBJ
20 LDA #22:LDY #0
30 STA $1500,Y
40 .END TEST
```

The pseudo-op .NH means no hex (causing the listing to change from hex to decimal), and .D means create a disk file containing the ML program which results from the assembly process.

You can add REM-like comments by using a semicolon. And you can turn the screen listing *off* with .NS, anytime. Turn it on or off as much as you want:

```
10 *= $0360
11 .NH
12 .D T.OBJECTPROGRAM
15 .NS
20 LDA #22:LDY #0; load A with 22, load Y with zero
30 STA $1500,Y
40 .END TEST
```

You turn on printer listings with .P and turn them off with .NP. However, for the .P pseudo-op to work, the .S screen listings pseudo-op must also be turned on. In other words, you cannot have listings sent to the printer without also having them listed on the screen at the same time. To have the ML stored into memory during assembly, use .O and turn off these POKEs to memory with .NO.

The pseudo-ops which turn the printer on and off; direct object code to disk, screen, and RAM; or switch between hex or decimal printout can be switched on and off within your source code wherever convenient. For example, you can turn on your printer anywhere within the program by inserting .P and turn it off anywhere with .NP. Among other things, this would allow you to specify that only a particular section of a large program be printed out. This can come in very handy if you're working on a 5000-byte program: you would have a long wait if you had to print out the whole thing.

Always put pseudo-ops on a line by themselves. Any other programming code can be put on a line in any fashion (divided by colons: LDA 15:STA 27:INY), but pseudo-ops should be the only thing on their lines. (The .BYTE pseudo-op is an exception—it can be on a multiple-statement line.)

100 .P .S	(wrong)
100 .P	(right)
110 .S	(right)

Here's a summary of the commands you can give LADS:

.P	Turn on printer listing of object code (.S must be activated).
.NP	Turn off printer listing of object code.
.O	Turn on POKEs to memory. Object code is stored into RAM *during* assembly.
.NO	Turn off POKEs to memory.
.D ***filename***	Open a file and store object code to disk during assembly (use no quotes around filename).
.FILE ***filename***	Link one source file to the next in a chain so that they will all assemble together as a single large source program (end the chain with .END pseudo-op).
.END ***filename***	Link the last source file to first source file in a chain. If you are assembling from a single file, give *its* filename as the .END so the assembler knows where to go for the second pass. Any source code must have .END as the last line in the program, whether the source code is con-

	tained within a single disk file or spread across a multiple-file chain.
.S	Turn on screen listing during assembly (required if you desire a hardcopy listing from a printer using the .P pseudo-op).
.NS	Turn off screen listing during assembly.
.H	Turn on hexadecimal output for screen or printer listing.
.NH	Turn off hexadecimal output for screen or printer listing. (As a result, the listings are in decimal.)
*=	Set program counter to new address.

A Stable Buffer

The pseudo-op *= is mainly useful when you want to create data tables. The subprogram Tables in LADS source code is an example. (A subprogram is one of the source code files which, when linked together, form an entire ML program.) You might want to create an ML program and locate its tables, equates, buffers, and messages at the high end of the ML program the way LADS does with its Tables subprogram. Since you don't know what the highest RAM address will be while you're writing the program, you can set *= to some address perhaps 4K above the starting address. This gives you space to write the program below the tables. The advantage of stable tables is that you can easily PEEK them and this greatly assists debugging. You'll always know exactly where buffers and variables are going to end up in memory after an assembly—regardless of the changes you make in the program.

Here's an example. Suppose you write:

```
10 *= $5000
20 STA BUFFER
30 *= $6000
40 BUFFER .BYTE 0 0 0 0 0 0 0 0 0 0 0 0 0 0
50 .END BUFFEREXAMPLE
```

This creates an ML instruction (STA buffer) at address $5000 (the starting address of this particular ML program), but places the buffer itself at $6000. When you add additional instructions after STA buffer, the location of the buffer itself will remain at address $6000. This means that you can write an entire program without having to worry that the location of the buffer is changing each time you add new instructions, new code. It's high enough so that it remains stable at $6000, and you can debug the program more easily. You can always check if something is being correctly sent into the buffer by just looking at $6000.

This fragment of code illustrates two other features of LADS. You can use the pseudo-op .BYTE to set aside some space in memory (the zeros above just make space to hold other things in a "buffer" during the execution of an ML program). You can also use .BYTE to define specific numbers in memory:

```
.BYTE 65 66 67 68
```

This would put these numbers (*you must always use decimal numbers with this pseudo-op*) into memory at the location of the .BYTE instruction. An easy way to create messages that you want to print to the screen is to use the .BYTE pseudo-op with quotes:

```
500 FIRSTLETTERS .BYTE "ABCD":.BYTE 0
```

Then, if you wanted to print this message, you could write:

```
2 *= $0360
5 LDY #0
10 LOOP LDA FIRSTLETTERS,Y
20 BEQ ENDMESSAGE
30 STA $0400,Y; location of screen RAM on Commodore 64
40 INY
50 JMP LOOP
60 ENDMESSAGE RTS; finished printout
500 FIRSTLETTERS .BYTE "ABCD:.BYTE 0
900 .END MESSAGETEST
```

Note that using the second set of quotes is optional with the .BYTE pseudo-op: You can use either **.BYTE "ABCD:.BYTE 0** or **.BYTE "ABCD":.BYTE 0**. To POKE numbers instead of characters, just leave out the quotes: **.BYTE 10 15 75**. And since these numeric values are being POKEd directly into bytes in memory, they cannot be larger than 255.

Labels

With LADS, or with other assemblers that permit labels, you need not refer to locations in memory or numeric values by using numbers. You can use labels.

In the example above, line 10 starts off with the word

LOOP. This means that you can use the word LOOP later on to refer to that location (see line 50). That's quite a convenience: The assembler remembers where the word LOOP is used and you need not refer to an actual memory *address*; you can refer to the label instead. Throughout this book, this kind of label is called a *PC-type* (for Program Counter) or *address-type* label.

The other type of label is defined is with an assembly convention called an *equate* (an equals sign). This is quite similar to the way that BASIC allows you to assign value to words—it's called "assigning variables" when you do it in BASIC. In ML, the = pseudo-op works pretty much the way the = sign does in BASIC. Here's an example:

```
5  *= $0360
10 SCREEN = $0400; the location of the 1st byte in RAM of the
   64 screen
20 HEARTSYMBOL = 83; the heart figure
30; -----------------------
40 START LDA HEARTSYMBOL; notice "START" (an address-
   type label)
50 STA SCREEN
60 RTS
```

Line 10 assigns the number $0400 (1024 decimal) to the word SCREEN. Anytime thereafter that you use the word SCREEN, LADS will substitute $0400 when it assembles your ML program. Line 20 "equates" the word HEARTSYMBOL to the number 83. So, when you LDA HEARTSYMBOL in line 40, the assembler will put an 83 into your program. (Notice that, like BASIC, LADS requires that equate labels be a single word. You couldn't use HEART SYMBOL, since that's two words.)

Line 30 is just a REMark. The semicolon tells the assembler that what follows on that line is to be ignored. Nevertheless, blank lines or graphic dividers like line 30 can help to visually separate subroutines, tables, and equates from your actual ML program. In this case, we've used line 30 to separate the section of the program which defines labels (lines 10–20) from the program proper (lines 40–60). All this makes it easier to read and understand your source code later.

Automatic Math

There are times when you will want to have LADS do addition for you. That's where the + pseudo-op comes in. If you write "label+1" you will add 1 to the value of the label. Here's how it works:

```
10 *= 864
20 MEMTOP = $34; top-of-memory pointer for 8032 PET.
30; ----------------------------
40 LDA #0:STA MEMTOP:LDA #$50:STA MEMTOP+1
```

Here we are putting a new location into the top-of-memory pointer which the computer uses to decide where it can store things. (Doing that could protect an ML program which resides above the address stored in this pointer.) Like all pointers, it uses two bytes. If we want to store $5000 into this pointer, we store the lower half (the least significant byte) into MEMTOP. We'll want to put the number $50 into the most significant byte of the pointer—but we don't want to waste time making a new label. It's just one higher in memory than MEMTOP. Hence, MEMTOP+1.

You'll also want to use the + pseudo-op command in constructions like this:

```
10 *= 864
15 SCREEN = $0400
17; ----------------------------
20 LDA #32; the blank character
30 LDA #0
40 START STA SCREEN,Y
50 STA SCREEN+256,Y
60 STA SCREEN+512,Y
70 STA SCREEN+768,Y
80 INY
90 BNE START
```

This is the fastest way to fill memory with a given byte. In this case we're clearing out the screen RAM by filling it with blanks. But it's easy to indicate multiples of 256 by just adding them to the label SCREEN.

A similar pseudo-op command is the #<. This refers to the least significant byte of a *label*. For example:

```
10 *= $0360
20 SCREEN = $8011
25 SCREENPOINTER = $FB
30 ;--------------------
```

40 LDA #<SCREEN; LSB (least significant byte of the label SCREEN, $11)
50 STA SCREENPOINTER

You'll find this technique used several times in the LADS source code. It puts the LSB (least signficant byte) or the MSB (most signficant byte) of a label into the LSB or MSB of a pointer. In the example above, we want to set up a pointer that will hold the address of the screen RAM. The pointer is called SCREENPOINTER and we want to put $11 (the LSB of SCREEN) into SCREENPOINTER. So, we extract the LSB of SCREEN in line 40 by using # combined with the less-than symbol. We would complete the job with the greater-than symbol to fetch the MSB: 60 LDA #>SCREEN. Notice that these symbols must be attached to the label; *no space is allowed*. For example, LDA #> SCREEN would create problems. This LSB or MSB extraction from a label is something you'll need to do from time to time. The #< and #> pseudo-ops do it for you.

Chained Files

It is sometimes convenient to create several source code subprograms, to break the ML program source code into several pieces. LADS source code is divided into a number of program files: Array, Equate, Math, Pseudo, etc. This way, you don't need to load the entire source code in the computer's memory when you just want to work on a particular part of it. It also allows you to assemble source code far larger than could fit into available RAM.

In the last line of each subprogram you want to link, you put the linking pseudo-op .FILE *NAME* (use no quotes) to tell the assembler which subprogram to assemble next. Subprograms, chained together in this fashion, will be treated as if they were one large program. The final subprogram in the chain ends with the special pseudo-op .END *NAME*, and this time the name is the filename of the first of the subprograms, the subprogram which begins the chain. It's like stringing pearls and then, at the end, tying thread so that the last pearl is next to the first, to form a necklace.

Remember that you always need to include the .END pseudo-op, even if you are assembling from a *single*, unlinked source code file. In such a case (where you're working with a solo file), you don't need the linking .FILE pseudo-op. Instead, refer the file to itself with .END *NAME* where you list the solo file's name. Here's an illustration of how three subprograms would be linked to form a complete program:

```
 5 *= 864
10; "FIRST"——first program in chain
20;its first line must contain the start address
30;----------
40 LDA #20
50 STA $0400
60 .FILE SECOND
```

Then you save this subprogram to disk (it's handy to let the first remark line in each subprogram identify the subprogram's filename):

SAVE "FIRST",8

Next you create SECOND, the next link in the chain. But here, you use no starting address; you enter no *= since only one start address is needed for any program:

```
10 ; "SECOND"
20 INY:INX:DEY:DEX
30 .FILE THIRD
```

SAVE"SECOND",8

Now write the final subprogram, ending it with the clasp pseudo-op .END *NAME* which links this last subprogram to the first:

```
10 ; "THIRD"
20 LDA #65:STA $0400
30 .END FIRST
```

SAVE "THIRD",8

When you want to assemble this chain, just type FIRST in the upper left-hand corner of the screen, SYS to LADS, and it will assemble the entire chain.

If you want the object code (the finished ML program) stored in the computer's memory during the LADS assembly, add this line to FIRST above:

35 .O

If you want to save the object code as an ML program on disk that can be later loaded into the computer and run, add this line to FIRST:

36 .D PROGRAMNAME

When LADS is finished assembling, there will be an ML program on disk called PROGRAMNAME. You can load it and SYS 864 (that was the start address we gave this program), and the newly assembled ML program will execute.

One additional pseudo-op is the #". It is sometimes useful when you want to load the Accumulator with a particular ASCII character and don't offhand recall the numerical value. The letter A is 65 in the ASCII code. If you LDA #65:STA SCREEN, you would store the letter A to the screen. But, for convenience, you can LDA #"A:STA SCREEN. You can, in other words, use the #" followed by the character itself rather than by its ASCII code number.

Rules for LADS

Here are the rules you need to follow when writing ML for LADS to assemble:

1. *In general, all equate labels (labels using an equals sign) should be defined at the start of your program.* While this isn't absolutely necessary for labels with numbers above 255 (see SCREEN in the example below), it *is* the best programming practice. It makes it easier for you to modify your programs and simplifies debugging. LADS itself locates all its equate labels in the subprogram Defs, the first subprogram in its chain of source code files.

What's more, it is *necessary* that any equate label with a value lower than 256 be defined before any ML mnemonics reference that label. So, to be on the safe side, just get into the habit of putting all equate labels at the very start of your programs:

```
10 *= 864
20 ARRAYPOINTER = $FB; (251 decimal), a zero page address
30 OTHERPOINTER = $FD; (253 decimal), another zero page
   address
40 ;-----------------------
50 LDY #0:LDA $41
60 STA ARRAYPOINTER,Y
70 SCREEN = $8000
```

Notice that it's permissible to define the label SCREEN anywhere in your program. It's not a zero page address. You do have to be careful, however, with zero page addresses (addresses lower than 255). So most ML programmers make it a habit to define all their equates at the start of their source code.

2. *Put only one pseudo-op on a line.* Don't use a colon to put two pseudo-ops on a single line:

```
10 *= 864
20 .O:.NH    (wrong)
30 .O        (right)
40 .NH       (right)
```

The main exception to this is the .BYTE pseudo-op. Sometimes it's useful to set up messages with a zero at their end to *delimit* them, to show that the message is complete. When you delimit messages with a zero, you don't need to know the length of the message; you just branch when you come upon a zero:

```
10 *= 864
20 SCREEN = $0364
30 ;----------------------
40 LDY #0
50 LOOP LDA MESSAGE,Y:BEQ END; loading a zero signals
   end of message.
60 STA SCREEN,Y:INY:  JMP LOOP; LADS ignores spaces after a
   colon.
70 ; --------- message area here ----------
80 MESSAGE .BYTE "PRINT THIS ON SCREEN":.BYTE 0
```

Any embedded pseudo-ops like + or = or #> can be used on multiple-statement lines. The only pseudo-ops which should be on a line by themselves are the I/O (input/output) instructions which direct communication to disk, screen, or printer, like .P, .S, .D, .END, etc.

Generally, it's important that you space things correctly. If you wrote:

```
SCREEN= 864
```

LADS would think that your label was *screen=* instead of *screen*. So you need that space between the label and the equals sign. Likewise, you need to put *a single space* between labels, mnemonics, and arguments:

```
LOOP LDA MESSAGE
```

Running them together will confuse LADS:

LOOPLDA MESSAGE

and

LOOP LDAMESSAGE

are wrong.

It's fine to have leading spaces following a colon, however. LADS will ignore those (see line 60 above). Also, spaces within remarks are ignored. In fact, LADS ignores anything following a semicolon (see line 70). However, the semicolon should come after anything you want assembled. You couldn't rearrange line 50 above by putting the BEQ END after the remark message. It would be ignored because it followed the semicolon.

When using the text form of .BYTE, it's up to you whether you use a close quote:

50 MESSAGE .BYTE "PRINT THIS" (right)
60 MESSAGE .BYTE "PRINT THIS (also right)

3. *The first character of any label must be a letter, not a number.* LADS knows when it comes upon a label because a number starts with a number; a label starts with a letter of the alphabet:

10 *= 864
20 LABEL = 255
30 LDA LABEL
40 LDA 255

Lines 30 and 40 accomplish the same thing and are correctly written. It would confuse LADS, however, if you wrote:

20 5LABEL = 255 (wrong)

since the number 5 at the start of the word *label* would signal the assembler that it had come upon a number, not a label. You can use numbers anywhere else in a label name—just don't put a number at the start of the name. Also avoid using symbols like # < > * and other punctuation, shifted letters, or graphics symbols within labels. Stick with ordinary alphanumerics:

10 5LABEL (wrong)
20 LABEL15 (right)
30 *LABEL* (wrong)

4. *Move the Program Counter forward, never backward.* The *= pseudo-op should be used to make space in memory. If

you set the PC below its current address, you would be writing over previously assembled code:

10 *= 864
20 LDA #15
30 *= 900 (right)

10 *= 864
20 LDA #15
30 *= 864 (wrong, you'll assemble right over the LDA #15)

Special Note to Tape Drive Users

LADS will assemble source code from disk or RAM memory. It is possible to use the assembler with a tape drive, using the RAM memory-based version (see Chapter 11). Of course, disk users can also assemble from RAM if they choose. But tape users must.

There is a restriction when using a tape drive as the outboard memory device. You cannot link files together, forming a large, chained source code listing. The reason for this is that LADS, like all sophisticated assemblers, makes two passes through the source code. This means that tape containing the source code would have to be rewound at the end of the first pass.

It would be possible, of course, to have LADS pause at the end of pass 1, announce that it's time to rewind the tape (see Atari notes below), and then, when you press a key, start reading the source code from the start of the tape. But this causes a second problem: The object code cannot then be stored to tape. A tape drive cannot simultaneously read and write.

The best way to use LADS with a tape drive is to assemble from source code in RAM memory and to use the .O (store object code to RAM pseudo-op). Then, when the finished object code is in RAM, use a monitor program like "Tinymon" or "Micromon" to save it to tape. If you have access to a disk drive, you could construct a version of LADS which automatically directs object code to tape during assembly using the .D pseudo-op.

Special Atari Notes

The Atari version of LADs is a complete programming environment. Unlike the Commodore and Apple versions of LADS, where you use the BASIC program editor to write and edit your source code, the Atari version has a special editor integrated into LADS itself. This is necessary because with Atari BASIC, you can only enter BASIC instructions. The line

10 *= $0600

is just as illegal as

10 PRIMT "NAME":INPPUT A#

Both are coolly received with an error message. This syntax checking is fine when working with BASIC, but prevents the standard BASIC editor from accepting and storing LADS source code. Once the decision was made to create an entirely new source code editor, LADS became a self-contained package. The BASIC cartridge is neither needed nor especially desired. Since LADS takes over the Atari, DOS is the only other program in memory, freeing up all the RAM ordinarily used by BASIC.

One note: If you'd rather use a word processor or other text editor to enter and edit your source code, you can, as long as your editor will send out numbered statements, in ASCII, ending with 155's (ATASCII carriage returns). Most Atari word processors conform to this; it you're not sure, experiment with a short source code program. Be sure to end each source line with a carriage return. You can then load the file into the LADS editor or assemble directly from disk with the LADS D:filename command.

Entering LADS

The object code for Atari LADS is typed in with the Atari version of MLX, a machine language entry editor. See Appendix C for details. After you've typed it in, you can save LADS to disk under the filename AUTORUN.SYS. This will cause LADS to load and automatically run when you turn on (boot) your computer and disk drive. LADS as assembled requires at least 40K of memory. If you have access to a 40K Atari, you can reassemble the source code to almost any memory location you want (see "Programming Atari LADS" in Chapter 11).

If you didn't save LADS as AUTORUN.SYS, you need to load it from the DOS menu, then use menu selection M and run it at address 8000. If you bought the LADS source/object code disk, LADS will automatically load and run when you insert the disk and turn on your system. LADS will then print its prompt, "LADS Ready." This indicates that LADS is ready to receive commands or source code.

Using the Editor

You enter your ML source code just as you do in BASIC. To start a new line, type a line number, then the text, followed by the RETURN key. To delete a line, type the line number by itself, then press RETURN. To insert a line between two existing lines, just give it a line number that falls between the two. For example, line 105 will end up between line 100 and 110.

The editor assumes that a line beginning with a line number should be stored as part of your source code. If your line starts with leading zeros, these leading zeros will be erased. As the editor reads the line you've entered, it converts lowercase to uppercase, and inverse video characters to normal ones. It will not convert characters within double quotes (SHIFT-2) or after a semicolon, which marks the start of a comment. This line:

0100 lda #"a":jmp ($fffc); FFFC is the reset vector

would become:

100 LDA #"a":JMP ($FFFC); FFFC is the reset vector

If there is no line number, the editor assumes you've entered an editor command. Note that if a command has any parameters after it, the command must be followed by a space.

Atari Editor Commands

LIST

LIST all by itself displays the entire source program. LIST 150 lists just line 150. LIST 110–160 shows all the lines between and including lines 110 through 160. If you want to list from a certain line number to the end of your program, just make the second line number very large, as in LIST 2000,9999. If you want to send a listing to the printer, use the SAVE command.

SAVE *device:filename*

SAVE works just like LIST, but sends the listing to the specified device with the given filename. To list the entire source code to the printer, use SAVE P:. Be sure to put a space between the command and the device. To LIST to cassette, use SAVE C:. When using disk, remember to use D:, for example, SAVE D:DEFS.SRC. We recommend that you do use an extender, such as .SRC (see .FILE below). Check the DOS manual for examples of legal filenames. You can also save a portion of the program. SAVE P:,100,150 would list lines 100 to 150 to the printer.

LOAD *device:filename*
Load will replace any source code in memory with that read from the specified device. LOAD C: reads from tape, LOAD D:DEFS.SRC or LOAD D2:INDISK.SRC from disk.

MERGE *device:filename*
Merge is used to combine two programs. MERGE works just like ENTER does in BASIC. Instead of the keyboard being used to accept text, the editor looks to the file for input. After all the lines have been entered, the editor restores keyboard control. MERGE does not just append one program to the other. If there is a line 150 in the program to be merged, it will replace line 150 in memory. Therefore, MERGE can replace selected lines, or add lines to the top or bottom of a program in memory. You can use SAVE to list to disk a part of a program, then use MERGE to add it to another program. You can have a whole disk full of commonly used routines, then use MERGE to combine the routines you need, speeding up the development of large ML programs.

DOS
If used with standard Atari DOS 2.0S, this command will load and run DUP.SYS, the DOS menu. Remember that DUP.SYS will erase any program in memory if MEM.SAV is not used. Now you can manipulate files and display the disk directory. The DOS command makes an indirect jump through the DOS vector, location $0A.

SYS *address*
Transfers control with a JSR to the decimal address following the SYS. Always put a space between SYS and the address. If the routine ends with a RTS, control will return to the LADS editor. If a BRK ($00) is encountered, the editor will also be reentered through the breakpoint, and the address where the BRK was found will be displayed.

LADS (optional device:filename)
Transfers control to the assembler. Although the editor merely manipulates text source code, it's as if all of LADS was just another editor command. When LADS takes control, the left margin is set to 0, to give a full 40-column width for printout.

The left margin reverts to 2 when the editor is reentered. If you give the filename, as in LADS D:DEFS.SRC, then LADS will assemble the given source code from disk. This is like Commodore LADS' default—assembling from disk. If you leave off the filename, LADS will behave as a RAM-based assembler, reading the current source code in memory and assembling it. Unlike Commodore or Apple LADS, where you change the source code and reassemble a separate version of LADS dedicated to RAM-based assembly, Atari LADS features both disk assembly and memory assembly in the same program, executing the appropriate code by checking RAMFLAG. For more information on this, see "Notes on the Structure of Atari LADS" in Chapter 11.

After an assembly is complete, or if you halt assembly by hitting the BREAK key, control will return to the editor.

Error Handling

Within the editor, any error will be displayed with Error - and the error number. This may be Error - 170 for file not found when you try to load a nonexistent file from the editor, or it may be an error returned from the assembler. Use your DOS or BASIC manual for a list of error numbers and error messages. Any illegal command or a command the editor can't understand will result in a Syntax Error.

Special Notes for Cassette Users

The filename for the cassette is C:. It is possible to assemble from cassette. When you see the .END, and hear the single tone, rewind the tape, press play, and then press any key to start the second pass. If you're using linked files, each file must link to the next with .FILE C:. The last source file should end with .END C:. Assembling from tape is a cumbersome affair in any case. It might be preferable for tape drive users to keep all source code in memory, then assemble to memory, using the cassette only to store and retrieve source code.

Pseudo-ops

All the pseudo-ops described above for the Commodore and Apple versions are fully operative in Atari LADS. A few usage notes follow:

.O This causes the assembler to POKE the object code into memory. Its converse is .NO. You must not overwrite the assembler, which uses memory from $8000 to approximately $9FFF. During assembly, the labels are stored below $8000, descending towards $7000. Only a very long program will need memory between $7000 and $8000 when it is assembled. Also avoid overwriting your source code, which starts at $2000 and works its way up.

A good location for very small routines is in page 6, $0600–$06FF. During assembly, all of page 5 will be corrupted. You can store your object code fairly safely at $5000 or $6000, assuming your source code in memory is not too long. You can break your source code into modules, which will link together with .FILE and .END (see below). If you remove all cartridges (or hold down OPTION when you turn on your machine, which removes BASIC on a 600XL or 800XL), there will be unused memory from $A000 to about $AFFF, less screen memory usage.

An alternative to .O is the .D pseudo-op, which stores the object code to disk. This entirely avoids any memory constraints. You can go to DOS and load the object code, then use the M. RUN AT ADDRESS option to execute and test it.

.D If storing object code to disk, be sure to use the D:, as in .D D:LADS.OBJ. Storing object code to tape is risky, since an excessively long leader may be written. Besides, there is no facility for loading cassette object files without a BASIC loader program. After the assembly is complete, you can go to the DOS menu and use menu selection L to load your program, then selection M to run it. Menu selection M. RUN AT ADDRESS requires a hexadecimal number without the dollar sign.

.P This assumes an 80-column printer. Remember to use it with .S if you want the assembly listing to also go to the printer. If the printer is not turned on, assembly will abort and you will be returned to the editor with an Error - 138.

.FILE Be sure to follow .FILE (or simply .F) with a space, then D:, followed by the filename. You may get occasional errors if you don't use an extender. It is recommended that you add the extender .SRC, as in VALDEC.SRC (SRC for SouRCe). For example, .FILE D:EVAL.SRC

.END Use this only at the end of the last file in a linked chain of source code. You can abbreviate it to .E. An example of proper usage is .END D:DEFS.SRC

Programming Aids

Following are two utility programs, written in BASIC. Program A-1 will renumber an Atari LADS source program. Just run it and follow the prompts. Program A-2 partially converts a file from the *Assembler Editor, EASMD,* or *MAC/65* assembler to the LADS syntax. It removes leading spaces after a line number, trailing spaces at the end of a line, and tucks comments right next to the operand fields. Into the DATA statements starting at 500, insert the filenames of the files you want converted. Be sure to make END the last item in the DATA statements. To use LADS to assemble code written for one of these other assemblers, you must complete the conversion yourself by adjusting the pseudo-ops. See the descriptions of the LADS pseudo-ops at the start of this appendix.

Program A-1. Atari LADS Renumber Utility

```
10 GRAPHICS 0:? ,"Renumber LADS":? ,"-------------"
20 DIM T$(20),F$(20),F2$(20),A$(120)
30 ? "Enter filename. Do not use D:":INPUT T$:F$="D:":F$(3)=T$
40 F2$="D:TEMP.":F2$(LEN(F2$)+1)=T$
50 TRAP 500:OPEN #1,4,0,F$:TRAP 40000
60 ? :? "We will renumber the entire file."
70 ? :? "What line number do you want the file":? "to start with?100{4 LEFT}";:INPUT T$:LNUM=VAL(T$)
80 ? :? "What step do you want between":? "each line?10{3 LEFT}";:INPUT T$:INCR=VAL(T$)
90 OPEN #2,8,0,"D:TEMP"
100 TRAP 150:INPUT #1,A$:Z=1
110 IF A$(Z,Z)<>" " THEN IF Z<LEN(A$) THEN Z=Z+1:GOTO 110
130 PRINT #2;LNUM;A$(Z):LNUM=LNUM+INCR
140 GOTO 100
150 IF PEEK(195)<>136 THEN 200
160 CLOSE #1:CLOSE #2:XIO 33,#1,0,0,F$:XIO 32,#1,0,0,F2$
170 ? :? "Finished!":END
200 ? "{BELL}Error - ";PEEK(195);" during renumber":END
500 ? "{BELL}Cannot open ";F$:? "Error - ";PEEK(195):END
```

Program A-2. Atari LADS File Converter Utility

```
3 GRAPHICS 0
4 DIM A$(100),T$(100),F$(20),F2$(50)
10 READ T$:? T$:F$="D:":F$(3)=T$:IF T$="END"
    THEN END
20 F2$="D:TEMP.":F2$(LEN(F2$)+1)=T$
100 OPEN #1,4,0,F$
110 OPEN #2,8,0,"D:TEMP"
130 TRAP 170:INPUT #1,A$:IF A$(1,1)="0" THEN
    A$=A$(2)
135 Z=LEN(A$)
140 IF A$(Z,Z)=" " THEN Z=Z-1:GOTO 140
142 A$=A$(1,Z):Z=1
144 IF A$(Z,Z)<>" " THEN Z=Z+1:GOTO 144
145 SZ=Z:Z=Z+1
146 IF A$(Z,Z)=" " THEN Z=Z+1:GOTO 146
147 T$=A$(Z):A$=A$(1,SZ):A$(SZ+1)=T$:Z=LEN(A
    $):IF T$(1,1)=";" THEN 169
150 IF A$(Z,Z)<>";" THEN Z=Z-1:IF Z THEN 150
152 SZ=Z:Z=Z-1:IF Z<0 THEN 169
154 IF A$(Z,Z)=" " THEN Z=Z-1:GOTO 154
156 T$=A$(SZ):A$=A$(1,Z):A$(Z+1)=T$
169 PRINT #2:A$:GOTO 130
170 CLOSE #1:CLOSE #2:XIO 33,#1,0,0,F$:XIO 3
    2,#1,0,0,F2$:GOTO 10
180 REM PUT YOUR FILENAMES HERE
190 REM E.G. DATA DEFS.SRC,EVAL.SRC,END
```

Special Apple Notes

Once you have typed in Apple LADS, you must BSAVE it to disk. The start address is $79FD and the length is $1674. To execute LADS you BRUN the binary file. After it loads and sets up its special wedge (see Chapter 11 for details on this wedge), you will be prompted with the BASIC prompt and a cursor. You can now type in your files and save them just as you would an Applesoft file. After saving the program to disk, you assemble it by typing:

ASM filename

Make sure you have a space between ASM and your filename. If you do not have the space, you will get a syntax error. With the wedge in, the BASIC tokenize routine does not execute, so you cannot type in BASIC programs after you BRUN LADS. Otherwise, all the features of Apple LADS operate as described under the general instructions at the start of this chapter.

LADS Object Code

LADS will run on the Commodore 64, VIC-20, PET/CBM, Atari, and Apple computers. If you have a Commodore or Atari you should use the "MLX" machine language editor to enter the object code for LADS. Complete instructions on how to enter the object code using MLX, as well as the MLX programs, can be found in Appendix C. PET/CBM owners may find it convenient to use their built-in machine language monitor to make the changes shown in Programs B-3a and B-3b. Apple users should use the Apple built-in monitor and enter the hex data found in Program B-5. Additional instructions for the use of LADS can be found in Appendix A, "How to Use LADS."

LADS is nearly 5K long, and for those who prefer not to type it in, it can be purchased on a disk by calling COMPUTE! Publications toll free at 1-800-334-0868. Be sure to state whether you want the Commodore, Atari, or Apple disk.

Program B-1. Commodore 64 LADS: MLX Format

```
11000 :169,000,160,048,153,113,123
11006 :062,136,208,250,169,248,047
11012 :133,176,133,055,141,135,009
11018 :062,169,042,133,177,133,214
11024 :056,141,136,062,169,001,069
11030 :141,157,062,185,000,004,059
11036 :201,032,240,012,176,003,180
11042 :024,105,064,153,150,061,079
11048 :200,076,025,043,153,150,175
11054 :061,200,185,000,004,201,185
11060 :032,208,226,136,132,183,201
11066 :032,248,049,032,184,050,141
11072 :169,000,141,119,062,032,075
11078 :104,051,173,138,062,208,038
11084 :063,032,133,056,169,230,247
11090 :032,210,255,169,076,032,088
11096 :210,255,169,065,032,210,005
11102 :255,169,068,032,210,255,059
11108 :169,083,032,210,255,032,113
11114 :133,056,173,128,062,208,098
11120 :011,169,068,133,251,169,145
11126 :061,133,252,032,219,050,097
11132 :173,122,062,133,253,141,240
11138 :115,062,173,123,062,133,030
```

```
11144 :254,141,116,062,032,225,198
11150 :255,173,119,062,240,003,226
11156 :076,168,046,032,104,051,113
11162 :169,000,141,127,062,141,026
11168 :137,062,172,138,062,208,171
11174 :003,076,198,043,140,158,016
11180 :062,173,156,062,240,012,109
11186 :032,142,056,032,063,056,047
11192 :032,103,056,032,063,056,014
11198 :173,149,062,240,003,032,081
11204 :059,055,076,106,050,173,203
11210 :114,062,240,023,201,003,077
11216 :208,114,169,001,141,114,187
11222 :062,173,071,061,208,104,125
11228 :169,008,024,109,113,062,193
11234 :141,113,062,076,185,045,080
11240 :173,138,062,240,057,160,038
11246 :255,200,185,068,061,240,223
11252 :046,153,150,061,201,032,119
11258 :208,243,200,185,068,061,191
11264 :201,061,208,003,076,233,014
11270 :045,162,000,142,158,062,063
11276 :138,153,150,061,185,068,255
11282 :061,240,008,157,068,061,101
11288 :232,200,076,016,044,157,237
11294 :068,061,076,198,043,032,252
11300 :130,048,032,036,048,076,150
11306 :198,043,173,089,061,201,039
11312 :064,176,006,173,090,061,106
11318 :238,137,062,073,128,141,065
11324 :120,062,032,207,048,076,093
11330 :197,044,160,000,140,127,222
11336 :062,173,071,061,201,032,160
11342 :240,003,076,071,047,185,188
11348 :072,061,201,065,144,003,118
11354 :238,127,062,153,089,061,052
11360 :200,185,072,061,240,022,108
11366 :153,089,061,201,065,144,047
11372 :003,238,127,062,200,185,155
11378 :072,061,240,006,153,089,223
11384 :061,076,112,044,136,140,177
11390 :126,062,173,128,062,208,117
11396 :064,173,127,062,208,162,160
11402 :169,089,133,251,169,061,242
11408 :133,252,160,000,173,089,183
11414 :061,201,048,176,007,024,155
11420 :230,251,144,002,230,252,241
11426 :177,251,240,016,201,041,064
11432 :240,012,201,044,240,008,145
```

```
11438 :201,032,240,004,200,076,159
11444 :162,044,072,152,072,169,083
11450 :000,145,251.032.219.050.115
11456 :104,168,104,145,251,173,113
11462 :089,061,201,035,240,063,119
11468 :201,040,240,023,173,114,227
11474 :062,201,008,240,055,201,209
11480 :003,208,113,169,008,024,229
11486 :109,113,062,141,113,062,054
11492 :076,185,045,172,126,062,126
11498 :185,089,061,201,041,240,027
11504 :016,173,114,062,201,001,039
11510 :208,009,169,016,024,109,013
11516 :113,062,141,113,062,173,148
11522 :114,062,201,006,240,083,196
11528 :076,126,045,076,153,045,017
11534 :173,138,062,208,003,076,162
11540 :126,045,056,173,122,062,092
11546 :229,253,072,173,123,062,170
11552 :229,254,176,014,201,255,137
11558 :240,004,104,076,010,048,008
11564 :104,016,012,076,062,045,103
11570 :240,004,104,076,010,048,020
11576 :104,016,003,076,010,048,057
11582 :056,233,002,141,122,062,166
11588 :169,000,141,123,062,076,127
11594 :126,045,172,126,062,136,229
11600 :185,089,061,201,044,208,100
11606 :004,200,076,242,046,173,059
11612 :113,062,201,076,208,003,243
11618 :076,135,045,173,123,062,200
11624 :208,085,173,114,062,201,179
11630 :006,176,013,201,002,240,236
11636 :009,169,004,024,109,113,032
11642 :062,141,113,062,032,130,150
11648 :055,032,168,055,076,233,235
11654 :045,172,126,062,185,089,045
11660 :061,201,041,208,005,169,057
11666 :108,141,113,062,076,227,105
11672 :045,173,090,061,201,034,244
11678 :208,006,173,091,061,141,070
11684 :122,062,173,114,062,201,130
11690 :001,208,209,169,008,024,021
11696 :109,113,062,141,113,062,008
11702 :076,126,045,032,130,055,134
11708 :076,233,045,173,114,062,123
11714 :201,002,240,004,201,007,081
11720 :208,012,173,113,062,024,024
11726 :105,008,141,113,062,076,199
```

```
11732 :227,045,201,006,176,009,108
11738 :173,113,062,024,105,012,195
11744 :141,113,062,032,130,055,245
11750 :032,194,055,173,138,062,116
11756 :208,003,076,165,046,173,139
11762 :156,062,208,003,076,165,144
11768 :046,173,158,062,208,062,189
11774 :173,152,062,240,042,169,068
11780 :020,056,229,211,141,139,032
11786 :062,032,204,255,162,004,217
11792 :032,201,255,172,139,062,109
11798 :016,005,160,002,076,031,056
11804 :046,169,032,032,210,255,004
11810 :136,208,250,032,204,255,095
11816 :162,001,032,198,255,169,089
11822 :020,133,211,169,150,133,094
11828 :251,169,061,133,252,032,182
11834 :046,056,169,030,056,229,132
11840 :211,141,140,062,169,030,049
11846 :133,211,173,152,062,240,017
11852 :031,032,204,255,162,004,252
11858 :032,201,255,172,140,062,176
11864 :240,010,048,008,169,032,083
11870 :032,210,255,136,208,250,161
11876 :032,204,255,162,001,032,018
11882 :198,255,032,155,056,173,207
11888 :150,062,240,017,201,001,015
11894 :208,005,169,060,076,127,251
11900 :046,169,062,032,210,255,130
11906 :032,192,056,173,159,062,036
11912 :240,019,032,063,056,169,203
11918 :059,032,210,255,169,000,099
11924 :133,251,169,002,133,252,064
11930 :032,046,056,032,133,056,253
11936 :173,119,062,208,003,076,033
11942 :140,043,173,138,062,208,162
11948 :027,238,138,062,173,115,157
11954 :062,133,253,173,116,062,209
11960 :133,254,032,204,255,169,207
11966 :001,032,195,255,032,248,185
11972 :049,076,061,043,032,204,149
11978 :255,169,001,032,195,255,085
11984 :169,002,032,195,255,173,010
11990 :152,062,240,021,032,204,157
11996 :255,162,004,032,201,255,105
12002 :169,013,032,210,255,032,169
12008 :204,255,169,004,032,195,067
12014 :255,076,116,164,185,089,099
12020 :061,201,088,240,101,136,047
```

```
12026 :136,185,089,061,201,041,195
12032 :208,003,076,231,044,173,223
12038 :123,062,208,015,173,114,189
12044 :062,201,002,240,082,201,032
12050 :005,240,078,201,001,240,015
12056 :122,173,114,062,201,001,185
12062 :208,012,173,113,062,024,110
12068 :105,024,141,113,062,076,045
12074 :227,045,173,114,062,201,096
12080 :005,240,008,169,049,032,039
12086 :218,047,076,071,047,173,174
12092 :113,062,024,105,028,141,021
12098 :113,062,076,227,045,032,109
12104 :167,056,032,142,056,169,182
12110 :087,133,251,169,062,133,145
12116 :252,032,046,056,032,133,123
12122 :056,076,233,045,173,123,028
12128 :062,208,068,173,114,062,015
12134 :201,002,208,012,169,016,198
12140 :024,109,113,062,141,113,158
12146 :062,076,126,045,201,001,113
12152 :240,016,201,003,240,012,064
12158 :201,005,240,008,169,050,031
12164 :032,218,047,076,071,047,111
12170 :169,020,024,109,113,062,123
12176 :141,113,062,185,091,061,029
12182 :201,089,208,010,173,113,176
12188 :062,201,182,240,003,076,152
12194 :025,047,076,126,045,173,142
12200 :114,062,201,002,208,012,255
12206 :169,024,024,109,113,062,163
12212 :141,113,062,076,227,045,076
12218 :201,001,240,016,201,003,080
12224 :240,012,201,005,240,008,130
12230 :169,051,032,218,047,076,023
12236 :071,047,169,028,024,109,140
12242 :113,062,141,113,062,076,009
12248 :227,045,141,139,062,140,202
12254 :141,062,142,140,062,169,170
12260 :186,032,210,255,104,170,161
12266 :104,168,152,072,138,072,172
12272 :152,032,205,189,173,139,106
12278 :062,172,141,062,174,140,229
12284 :062,096,160,000,152,153,107
12290 :068,061,200,192,080,208,043
12296 :248,096,032,133,056,032,093
12302 :167,056,032,142,056,169,124
12308 :198,133,251,169,061,133,197
12314 :252,032,046,056,032,133,065
```

```
12320 :056,076,126,045,160,255,238
12326 :200,185,068,061,240,086,110
12332 :201,032,208,246,200,200,107
12338 :140,132,062,056,165,176,013
12344 :237,132,062,133,176,165,193
12350 :177,233,000,133,177,160,174
12356 :000,185,068,061,073,128,071
12362 :145,176,200,185,068,061,141
12368 :201,032,240,005,145,176,111
12374 :076,076,048,200,185,068,227
12380 :061,201,061,240,059,136,082
12386 :165,253,145,176,200,165,178
12392 :254,145,176,174,132,062,023
12398 :202,160,000,189,068,061,022
12404 :240,008,153,068,061,232,110
12410 :200,076,113,048,153,068,012
12416 :061,096,032,133,056,032,026
12422 :142,056,032,167,056,169,244
12428 :255,133,251,169,061,133,118
12434 :252,032,046,056,032,133,185
12440 :056,076,202,048,136,140,042
12446 :133,062,173,128,062,208,156
12452 :023,200,200,200,140,121,024
12458 :062,169,068,024,109,121,211
12464 :062,133,251,169,061,105,189
12470 :000,133,252,032,219,050,100
12476 :172,133,062,173,122,062,144
12482 :145,176,173,123,062,200,049
12488 :145,176,104,104,076,233,014
12494 :045,173,135,062,133,178,164
12500 :173,136,062,133,179,032,159
12506 :221,049,169,255,141,155,184
12512 :062,056,165,176,229,178,066
12518 :165,177,229,179,176,099,231
12524 :162,000,056,165,178,233,006
12530 :002,133,178,165,179,233,108
12536 :000,133,179,160,000,177,129
12542 :178,048,012,165,178,208,019
12548 :002,198,179,198,178,232,223
12554 :076,253,048,165,178,141,103
12560 :142,062,165,179,141,143,080
12566 :062,177,178,205,120,062,058
12572 :240,003,076,063,049,232,179
12578 :142,121,062,162,001,173,183
12584 :137,062,240,004,200,032,203
12590 :221,049,200,185,089,061,083
12596 :240,083,201,048,144,079,079
12602 :232,209,178,240,241,173,051
12608 :142,062,133,178,173,143,127
```

```
12614 :062,133,179,032,221,049,234
12620 :076,225,048,173,155,062,047
12626 :048,001,096,173,138,062,088
12632 :208,002,240,023,032,167,248
12638 :056,032,142,056,032,063,219
12644 :056,169,239,133,251,169,093
12650 :061,133,252,032,046,056,174
12656 :032,133,056,104,104,173,202
12662 :113,062,041,031,201,016,070
12668 :240,008,173,150,062,208,197
12674 :003,076,227,045,076,126,171
12680 :045,236,121,062,240,003,075
12686 :076,063,049,238,155,062,017
12692 :240,003,032,230,049,172,106
12698 :121,062,173,137,062,240,181
12704 :001,200,177,178,141,122,211
12710 :062,200,177,178,141,123,023
12716 :062,173,150,062,240,010,101
12722 :201,002,208,030,173,123,147
12728 :062,141,122,062,173,149,125
12734 :062,240,019,024,173,147,087
12740 :062,109,122,062,141,122,046
12746 :062,173,148,062,109,123,111
12752 :062,141,123,062,173,138,139
12758 :062,208,001,096,076,063,208
12764 :049,165,178,208,002,198,252
12770 :179,198,178,096,032,167,052
12776 :056,169,057,133,251,169,043
12782 :062,133,252,032,046,056,051
12788 :032,133,056,096,032,204,029
12794 :255,169,001,032,195,255,133
12800 :169,001,133,184,169,008,152
12806 :133,186,169,003,133,185,047
12812 :169,150,133,187,169,061,113
12818 :133,188,032,193,225,096,117
12824 :169,002,133,184,169,008,177
12830 :133,186,169,002,133,185,070
12836 :169,150,133,187,169,061,137
12842 :133,188,032,193,225,032,077
12848 :204,255,096,169,004,133,141
12854 :184,169,004,133,186,169,131
12860 :000,133,183,032,193,225,058
12866 :032,204,255,096,032,204,121
12872 :255,169,000,133,147,133,141
12878 :144,169,008,133,186,169,119
12884 :150,133,187,169,061,133,149
12890 :188,032,117,225,032,204,120
12896 :255,165,043,133,167,165,000
12902 :044,133,168,096,160,000,191
```

```
12908 :162,255,232,185,028,060,006
12914 :205,068,061,240,010,200,130
12920 :200,200,224,057,208,240,225
12926 :076,232,043,200,185,028,122
12932 :060,205,069,061,240,006,005
12938 :200,200,208,224,240,238,168
12944 :200,185,028,060,205,070,124
12950 :061,240,005,200,208,210,050
12956 :240,224,173,071,061,201,102
12962 :032,240,004,201,000,208,079
12968 :213,189,196,060,141,114,057
12974 :062,188,252,060,140,113,221
12980 :062,076,201,043,162,001,213
12986 :032,198,255,162,006,032,103
12992 :228,255,202,208,250,032,087
12998 :228,255,201,172,240,014,028
13004 :169,181,133,251,169,061,144
13010 :133,252,032,046,056,076,037
13016 :200,046,096,160,000,177,127
13022 :251,240,004,200,076,221,190
13028 :050,140,178,061,136,169,194
13034 :000,141,122,062,141,123,055
13040 :062,162,001,142,140,062,041
13046 :177,251,041,015,141,176,023
13052 :061,141,179,061,169,000,095
13058 :141,177,061,141,180,061,251
13064 :202,240,018,032,045,051,084
13070 :173,176,061,141,179,061,037
13076 :173,177,061,141,180,061,045
13082 :076,008,051,238,140,062,089
13088 :174,140,062,032,084,051,063
13094 :136,206,178,061,208,202,005
13100 :096,024,014,176,061,046,205
13106 :177,061,014,176,061,046,073
13112 :177,061,024,173,179,061,219
13118 :109,176,061,141,176,061,018
13124 :173,180,061,109,177,061,061
13130 :141,177,061,014,176,061,192
13136 :046,177,061,096,024,173,145
13142 :176,061,109,122,062,141,245
13148 :122,062,173,177,061,109,028
13154 :123,062,141,123,062,096,193
13160 :032,254,047,160,000,140,225
13166 :128,062,140,159,062,140,033
13172 :150,062,140,149,062,173,084
13178 :154,062,208,012,032,228,050
13184 :255,141,117,062,032,228,195
13190 :255,141,118,062,032,228,202
13196 :255,208,008,032,231,052,158
```

```
13202 :104,104,076,140,043,201,046
13208 :032,240,239,076,166,051,188
13214 :032,228,255,208,003,076,192
13220 :231,052,201,058,208,003,149
13226 :076,080,052,201,059,208,078
13232 :115,140,139,062,173,152,189
13238 :062,240,085,141,159,062,163
13244 :173,139,062,240,006,032,072
13250 :238,051,076,022,052,032,153
13256 :228,255,240,014,201,127,241
13262 :144,003,032,094,052,153,172
13268 :068,061,200,076,199,051,099
13274 :032,142,056,032,063,056,087
13280 :032,155,056,032,133,056,176
13286 :169,000,141,139,062,076,049
13292 :022,052,141,159,062,141,045
13298 :139,062,160,000,032,228,095
13304 :255,208,007,153,000,002,105
13310 :172,139,062,096,016,003,230
13316 :032,022,055,153,000,002,012
13322 :200,076,246,051,032,228,075
13328 :255,240,003,076,014,052,144
13334 :032,231,052,173,139,062,199
13340 :208,005,104,104,076,140,153
13346 :043,096,201,177,240,091,114
13352 :201,179,240,095,201,170,102
13358 :208,003,238,149,062,201,139
13364 :172,208,003,076,147,052,198
13370 :201,046,240,022,201,036,036
13376 :240,021,201,127,144,003,032
13382 :032,094,052,153,068,061,018
13388 :200,076,158,051,141,154,088
13394 :062,096,076,139,053,153,149
13400 :068,061,200,076,006,053,040
13406 :056,233,127,141,131,062,076
13412 :162,255,206,131,062,240,132
13418 :008,232,189,158,160,016,101
13424 :250,048,243,232,189,158,208
13430 :160,048,007,153,068,061,103
13436 :200,076,115,052,041,127,223
13442 :096,169,002,141,150,062,238
13448 :076,158,051,169,001,141,220
13454 :150,062,076,158,051,032,159
13460 :158,051,173,138,062,240,202
13466 :011,169,042,032,210,255,105
13472 :032,155,056,032,133,056,112
13478 :173,128,062,208,032,160,161
13484 :000,185,068,061,201,032,207
13490 :240,004,200,076,173,052,155
```

```
13496 :200,132,251,169,068,024,004
13502 :101,251,133,251,169,061,132
13508 :105,000,133,252,032,219,169
13514 :050,173,138,062,240,008,105
13520 :173,151,062,240,003,032,101
13526 :213,054,173,122,062,133,203
13532 :253,173,123,062,133,254,194
13538 :104,104,076,140,043,153,078
13544 :068,061,200,192,080,208,017
13550 :248,153,068,061,032,228,004
13556 :255,032,228,255,240,006,236
13562 :169,000,141,154,062,096,104
13568 :169,001,141,119,062,096,076
13574 :162,000,032,228,255,240,155
13580 :044,201,058,240,040,201,028
13586 :032,240,243,201,059,240,009
13592 :032,201,044,240,015,201,245
13598 :041,240,011,157,129,061,157
13604 :232,153,068,061,200,076,058
13610 :008,053,142,129,062,153,077
13616 :068,061,200,032,077,053,027
13622 :076,158,051,141,139,062,169
13628 :169,000,142,129,062,153,203
13634 :068,061,032,077,053,173,018
13640 :139,062,076,161,051,169,218
13646 :000,141,122,062,141,123,155
13652 :062,170,014,122,062,046,048
13658 :123,062,014,122,062,046,007
13664 :123,062,014,122,062,046,013
13670 :123,062,014,122,062,046,019
13676 :123,062,189,129,061,201,105
13682 :065,144,002,233,007,041,094
13688 :015,013,122,062,141,122,083
13694 :062,232,236,129,062,208,031
13700 :209,238,128,062,169,001,171
13706 :096,192,000,240,014,174,086
13712 :138,062,208,009,072,152,017
13718 :072,032,036,048,104,168,098
13724 :104,153,068,061,200,032,006
13730 :228,255,153,068,061,200,103
13736 :201,066,208,104,169,000,148
13742 :141,144,062,173,138,062,126
13748 :240,023,140,141,062,173,191
13754 :156,062,240,015,032,142,065
13760 :056,032,063,056,032,103,022
13766 :056,032,063,056,172,141,206
13772 :062,032,228,255,153,068,234
13778 :061,200,201,032,208,245,133
13784 :032,228,255,153,068,061,245
```

```
13790 :200,201,034,208,069,032,198
13796 :228,255,208,003,076,186,160
13802 :054,201,058,208,003,076,066
13808 :189,054,201,059,208,012,195
13814 :032,238,051,174,152,062,187
13820 :142,159,062,076,186,054,163
13826 :201,034,208,003,076,227,239
13832 :053,174,138,062,208,009,140
13838 :032,032,056,076,227,053,234
13844 :076,139,057,153,068,061,062
13850 :170,140,141,062,032,248,051
13856 :055,172,141,062,200,076,226
13862 :227,053,162,000,142,145,255
13868 :062,157,169,061,232,173,130
13874 :145,062,208,117,032,228,074
13880 :255,240,067,201,058,240,093
13886 :063,201,059,208,012,032,125
13892 :238,051,174,152,062,142,119
13898 :159,062,076,126,054,141,180
13904 :109,061,173,138,062,208,063
13910 :013,173,109,061,201,032,163
13916 :208,211,032,032,056,076,195
13922 :049,054,173,109,061,153,185
13928 :068,061,200,201,032,240,138
13934 :024,201,000,240,020,201,028
13940 :058,240,016,157,169,061,049
13946 :232,076,049,054,238,145,148
13952 :062,141,110,061,076,079,145
13958 :054,169,169,133,251,169,055
13964 :061,133,252,140,141,062,161
13970 :032,219,050,174,122,062,037
13976 :032,248,055,172,141,062,094
13982 :169,000,162,005,157,169,052
13988 :061,202,208,250,076,049,242
13994 :054,173,138,062,208,003,040
14000 :032,032,056,173,110,061,128
14006 :201,058,240,003,032,231,179
14012 :052,141,154,062,238,158,225
14018 :062,104,104,173,138,062,069
14024 :240,008,173,156,062,240,055
14030 :003,076,108,046,076,140,143
14036 :043,173,138,062,201,002,063
14042 :208,001,096,032,204,255,246
14048 :162,002,032,201,255,056,164
14054 :173,122,062,229,253,141,186
14060 :120,062,173,123,062,229,237
14066 :254,141,121,062,169,000,221
14072 :032,210,255,173,120,062,076
14078 :208,003,206,121,062,206,036
```

```
14084 :120,Ø62,2Ø8,238,173,121,158
1409Ø :Ø62,2Ø8,233,Ø32,2Ø4,255,236
14096 :162,ØØ1,Ø32,198,255,Ø96,248
141Ø2 :Ø56,233,127,141,131,Ø62,ØØ4
141Ø8 :162,255,2Ø6,131,Ø62,24Ø,Ø6Ø
14114 :ØØ8,232,189,158,16Ø,Ø16,Ø29
1412Ø :25Ø,Ø48,243,232,189,158,136
14126 :16Ø,Ø48,ØØ7,153,ØØØ,ØØ2,16Ø
14132 :2ØØ,Ø76,Ø43,Ø55,Ø41,127,Ø82
14138 :Ø96,16Ø,ØØØ,162,ØØØ,185,149
14144 :Ø68,Ø61,2Ø1,Ø43,24Ø,ØØ4,169
1415Ø :2ØØ,Ø76,Ø63,Ø55,2ØØ,185,Ø81
14156 :Ø68,Ø61,Ø32,Ø9Ø,Ø55,176,Ø46
14162 :Ø18,157,129,Ø61,232,Ø76,243
14168 :Ø74,Ø55,2Ø1,Ø58,176,ØØ6,146
14174 :Ø56,233,Ø48,Ø56,233,2Ø8,16Ø
1418Ø :Ø96,169,ØØØ,157,129,Ø61,2ØØ
14186 :169,129,133,251,169,Ø61,25Ø
14192 :133,252,Ø32,219,Ø5Ø,173,2Ø3
14198 :122,Ø62,141,147,Ø62,173,Ø57
142Ø4 :123,Ø62,141,148,Ø62,Ø96,244
1421Ø :173,138,Ø62,2Ø8,ØØ4,Ø32,235
14216 :Ø32,Ø56,Ø96,173,156,Ø62,199
14222 :24Ø,Ø17,Ø32,2Ø4,255,162,Ø28
14228 :ØØ1,Ø32,198,255,174,113,153
14234 :Ø62,Ø32,Ø72,Ø56,Ø32,Ø63,215
1424Ø :Ø56,174,113,Ø62,Ø32,248,Ø77
14246 :Ø55,Ø96,173,138,Ø62,2Ø8,13Ø
14252 :ØØ4,Ø32,Ø32,Ø56,Ø96,173,Ø53
14258 :156,Ø62,24Ø,ØØ6,174,122,17Ø
14264 :Ø62,Ø32,Ø72,Ø56,174,122,19Ø
1427Ø :Ø62,Ø76,248,Ø55,173,138,174
14276 :Ø62,2Ø8,ØØ7,Ø32,Ø32,Ø56,Ø81
14282 :Ø32,Ø32,Ø56,Ø96,173,156,235
14288 :Ø62,24Ø,ØØ6,174,122,Ø62,1Ø6
14294 :Ø32,Ø72,Ø56,174,122,Ø62,22Ø
143ØØ :Ø32,248,Ø55,173,156,Ø62,178
143Ø6 :24Ø,Ø14,173,157,Ø62,24Ø,Ø88
14312 :ØØ3,Ø32,Ø63,Ø56,174,123,171
14318 :Ø62,Ø32,Ø72,Ø56,174,123,245
14324 :Ø62,Ø76,248,Ø55,142,121,18Ø
1433Ø :Ø62,173,153,Ø62,24Ø,ØØ5,177
14336 :16Ø,ØØØ,138,145,253,173,1Ø1
14342 :151,Ø62,24Ø,Ø22,Ø32,2Ø4,2Ø5
14348 :255,162,ØØ2,Ø32,2Ø1,255,151
14354 :173,121,Ø62,Ø32,21Ø,255,1Ø3
1436Ø :Ø32,2Ø4,255,162,ØØ1,Ø32,198
14366 :198,255,Ø24,169,ØØ1,1Ø1,Ø1Ø
14372 :253,133,253,169,ØØØ,1Ø1,177
```

```
14378 :254,133,254,096,160,000,171
14384 :177,251,240,010,032,210,200
14390 :255,032,186,056,200,076,091
14396 :048,056,096,169,032,032,237
14402 :210,255,032,186,056,096,133
14408 :142,140,062,173,157,062,040
14414 :240,011,138,032,114,057,158
14420 :032,227,056,174,140,062,007
14426 :096,169,000,032,205,189,013
14432 :032,227,056,174,140,062,019
14438 :096,173,157,062,240,014,076
14444 :165,254,032,114,057,165,127
14450 :253,032,114,057,032,022,112
14456 :057,096,166,253,165,254,087
14462 :032,205,189,032,022,057,151
14468 :096,169,013,032,210,255,139
14474 :032,186,056,096,174,117,031
14480 :062,173,118,062,032,205,028
14486 :189,032,076,057,096,169,001
14492 :068,133,251,169,061,133,203
14498 :252,032,046,056,096,169,045
14504 :007,032,210,255,169,018,091
14510 :032,210,255,032,155,056,146
14516 :169,013,032,210,255,096,187
14522 :174,138,062,208,001,096,097
14528 :174,152,062,208,001,096,117
14534 :141,139,062,032,204,255,007
14540 :162,004,032,201,255,173,007
14546 :139,062,032,210,255,032,172
14552 :204,255,162,001,032,198,044
14558 :255,173,139,062,096,174,097
14564 :138,062,208,001,096,174,139
14570 :152,062,208,001,096,032,017
14576 :204,255,162,004,032,201,074
14582 :255,173,157,062,240,009,118
14588 :173,140,062,032,114,057,062
14594 :076,013,057,169,000,174,235
14600 :140,062,032,205,189,032,156
14606 :204,255,162,001,032,198,098
14612 :255,096,174,138,062,208,185
14618 :001,096,174,152,062,208,207
14624 :001,096,032,204,255,162,014
14630 :004,032,201,255,174,157,093
14636 :062,240,013,165,254,032,042
14642 :114,057,165,253,032,114,017
14648 :057,076,067,057,165,254,220
14654 :166,253,032,205,189,032,171
14660 :204,255,162,001,032,198,152
14666 :255,096,174,138,062,208,239
```

```
14672 :001,096,174,152,062,208,005
14678 :001,096,032,204,255,162,068
14684 :004,032,201,255,173,118,107
14690 :062,174,117,062,032,205,238
14696 :189,032,204,255,162,001,179
14702 :032,198,255,096,072,041,036
14708 :015,168,185,052,061,170,255
14714 :104,074,074,074,074,168,178
14720 :185,052,061,032,210,255,155
14726 :138,032,210,255,096,201,042
14732 :070,208,008,032,238,057,241
14738 :104,104,076,140,043,201,046
14744 :128,208,006,032,071,058,143
14750 :076,146,057,201,068,208,146
14756 :003,076,127,058,201,080,197
14762 :208,003,076,244,058,201,192
14768 :078,208,003,076,053,059,141
14774 :201,079,208,003,076,032,013
14780 :059,201,083,208,003,076,050
14786 :237,059,201,072,208,003,206
14792 :076,007,060,153,068,061,113
14798 :032,142,056,032,063,056,075
14804 :032,103,056,032,167,056,146
14810 :032,155,056,169,087,133,082
14816 :251,169,062,133,252,032,099
14822 :046,056,032,133,056,076,117
14828 :007,059,032,228,255,201,250
14834 :032,240,003,076,238,057,120
14840 :160,000,032,228,255,201,100
14846 :000,240,014,201,127,144,212
14852 :003,032,094,052,153,068,150
14858 :061,200,076,250,057,132,018
14864 :183,160,000,185,068,061,161
14870 :240,007,153,150,061,200,065
14876 :076,019,058,173,138,062,042
14882 :208,006,032,103,056,032,215
14888 :063,056,032,155,056,032,178
14894 :133,056,032,248,049,162,214
14900 :001,032,198,255,032,228,030
14906 :255,032,228,255,032,231,067
14912 :052,162,000,142,119,062,089
14918 :096,169,046,032,210,255,110
14924 :169,069,032,210,255,169,212
14930 :078,032,210,255,169,068,126
14936 :032,210,255,169,032,032,050
14942 :210,255,032,228,255,032,082
14948 :248,057,173,138,062,240,250
14954 :003,238,119,062,238,138,136
14960 :062,173,115,062,133,253,142
```

```
14966 :173,116,Ø62,133,254,Ø32,12Ø
14972 :1Ø4,Ø51,Ø96,173,138,Ø62,236
14978 :24Ø,Ø3Ø,Ø32,228,255,153,Ø44
14984 :Ø68,Ø61,16Ø,ØØØ,Ø32,228,173
1499Ø :255,24Ø,Ø2Ø,2Ø1,127,144,1Ø5
14996 :ØØ3,Ø32,Ø94,Ø52,153,Ø68,Ø38
15ØØ2 :Ø61,153,15Ø,Ø61,2ØØ,Ø76,Ø87
15ØØ8 :14Ø,Ø58,Ø76,ØØ7,Ø59,169,157
15Ø14 :Ø44,153,15Ø,Ø61,2ØØ,169,175
15Ø2Ø :Ø8Ø,153,15Ø,Ø61,2ØØ,169,217
15Ø26 :Ø44,153,15Ø,Ø61,2ØØ,169,187
15Ø32 :Ø87,153,15Ø,Ø61,2ØØ,132,199
15Ø38 :183,Ø32,155,Ø56,Ø32,133,Ø13
15Ø44 :Ø56,238,151,Ø62,Ø32,Ø24,247
15Ø5Ø :Ø5Ø,162,ØØ2,Ø32,2Ø1,255,136
15Ø56 :173,115,Ø62,Ø32,21Ø,255,Ø31
15Ø62 :173,116,Ø62,Ø32,21Ø,255,Ø38
15Ø68 :Ø32,2Ø4,255,162,ØØ1,Ø32,138
15Ø74 :198,255,Ø32,2Ø5,Ø59,Ø32,239
15Ø8Ø :231,Ø52,1Ø4,1Ø4,162,ØØØ,117
15Ø86 :142,119,Ø62,Ø76,14Ø,Ø43,Ø52
15Ø92 :173,138,Ø62,24Ø,Ø14,Ø32,135
15Ø98 :Ø51,Ø5Ø,238,152,Ø62,Ø32,Ø67
151Ø4 :2Ø4,255,162,ØØ1,Ø32,198,Ø84
1511Ø :255,Ø32,228,255,24Ø,ØØ7,255
15116 :2Ø1,Ø58,24Ø,ØØ6,Ø76,ØØ7,Ø88
15122 :Ø59,Ø32,231,Ø52,1Ø4,1Ø4,Ø88
15128 :162,ØØØ,142,119,Ø62,Ø76,Ø73
15134 :14Ø,Ø43,169,Ø46,Ø32,21Ø,158
1514Ø :255,169,Ø79,Ø32,21Ø,255,Ø12
15146 :Ø32,133,Ø56,169,ØØ1,141,Ø62
15152 :153,Ø62,Ø76,ØØ7,Ø59,173,Ø66
15158 :138,Ø62,24Ø,2Ø5,Ø32,228,191
15164 :255,2Ø1,Ø8Ø,24Ø,Ø12,2Ø1,Ø25
1517Ø :Ø79,24Ø,Ø58,2Ø1,Ø83,24Ø,199
15176 :1Ø6,2Ø1,Ø72,24Ø,Ø76,169,168
15182 :Ø46,Ø32,21Ø,255,169,Ø78,1ØØ
15188 :Ø32,21Ø,255,169,Ø8Ø,Ø32,Ø94
15194 :21Ø,255,Ø32,133,Ø56,2Ø6,214
152ØØ :152,Ø62,Ø32,2Ø4,255,162,195
152Ø6 :ØØ4,Ø32,2Ø1,255,169,Ø13,ØØ8
15212 :Ø32,21Ø,255,169,ØØ4,Ø32,Ø42
15218 :195,255,Ø32,2Ø4,255,162,193
15224 :ØØ1,Ø32,198,255,Ø76,ØØ7,177
1523Ø :Ø59,169,Ø46,Ø32,21Ø,255,129
15236 :169,Ø78,Ø32,21Ø,255,169,Ø21
15242 :Ø79,Ø32,21Ø,255,Ø32,133,111
15248 :Ø56,169,ØØØ,141,153,Ø62,213
15254 :Ø76,ØØ7,Ø59,169,Ø46,Ø32,Ø27
```

```
15260 :210,255,169,078,032,210,086
15266 :255,169,072,032,210,255,131
15272 :032,133,056,169,000,141,187
15278 :157,062,076,007,059,169,192
15284 :046,032,210,255,169,078,202
15290 :032,210,255,169,083,032,199
15296 :210,255,032,133,056,169,023
15302 :000,141,156,062,076,007,128
15308 :059,166,144,208,001,096,110
15314 :169,000,032,072,056,032,059
15320 :063,056,169,021,133,251,141
15326 :169,062,133,252,032,167,013
15332 :056,032,046,056,104,104,114
15338 :076,200,046,169,046,032,035
15344 :210,255,169,083,032,210,175
15350 :255,032,133,056,173,138,009
15356 :062,240,005,169,001,141,102
15362 :156,062,076,007,059,169,019
15368 :046,032,210,255,169,072,024
15374 :032,210,255,032,133,056,220
15380 :169,001,141,157,062,076,114
15386 :007,059,076,068,065,076,121
15392 :068,089,074,083,082,082,254
15398 :084,083,066,067,083,066,231
15404 :069,081,066,067,067,067,205
15410 :077,080,066,078,069,076,240
15416 :068,088,074,077,080,083,014
15422 :084,065,083,084,089,083,038
15428 :084,088,073,078,089,068,036
15434 :069,089,068,069,088,068,013
15440 :069,067,073,078,088,073,016
15446 :078,067,067,080,089,067,022
15452 :080,088,083,066,067,083,047
15458 :069,067,065,068,067,067,245
15464 :076,067,084,065,088,084,056
15470 :065,089,084,088,065,084,073
15476 :089,065,080,072,065,080,055
15482 :076,065,066,082,075,066,040
15488 :077,073,066,080,076,065,053
15494 :078,068,079,082,065,069,063
15500 :079,082,066,073,084,066,078
15506 :086,067,066,086,083,082,104
15512 :079,076,082,079,082,076,114
15518 :083,082,067,076,068,067,089
15524 :076,073,065,083,076,080,105
15530 :072,080,080,076,080,082,128
15536 :084,073,083,069,068,083,124
15542 :069,073,084,083,088,084,151
15548 :088,083,067,076,086,078,154
```

```
15554 :079,080,001,005,009,000,112
15560 :008,008,008,001,008,005,238
15566 :006,001,002,002,000,000,217
15572 :000,002,000,002,004,004,224
15578 :001,000,001,000,000,000,220
15584 :000,000,000,000,000,008,232
15590 :008,001,001,001,007,008,000
15596 :008,003,003,003,000,000,253
15602 :003,000,000,000,000,000,245
15608 :000,000,000,000,161,160,057
15614 :032,096,176,240,144,193,111
15620 :208,162,076,129,132,134,077
15626 :200,136,202,198,232,230,184
15632 :192,224,225,056,097,024,066
15638 :170,168,138,152,072,104,058
15644 :000,048,016,033,001,065,191
15650 :036,080,112,034,098,066,204
15656 :216,088,002,008,040,064,202
15662 :248,120,186,154,184,234,148
15668 :048,049,050,051,052,053,099
15674 :054,055,056,057,065,066,155
15680 :067,068,069,070,000,000,082
15686 :000,000,000,000,000,000,070
15692 :000,000,000,000,000,000,076
15698 :000,000,000,000,000,000,082
15704 :000,000,000,000,000,000,088
15710 :000,000,000,000,000,000,094
15716 :000,000,000,000,000,000,100
15722 :000,000,000,000,000,000,106
15728 :000,000,000,000,000,000,112
15734 :000,000,000,000,000,000,118
15740 :000,000,000,000,000,000,124
15746 :000,000,000,000,000,000,130
15752 :000,000,000,000,000,000,136
15758 :000,000,000,000,000,000,142
15764 :000,000,000,000,000,000,148
15770 :000,000,000,000,000,000,154
15776 :000,000,000,000,000,000,160
15782 :000,000,000,000,000,000,166
15788 :000,000,000,000,000,000,172
15794 :000,000,000,078,079,032,111
15800 :083,084,065,082,084,032,102
15806 :065,068,068,082,069,083,113
15812 :083,000,045,045,045,045,203
15818 :045,045,045,045,045,045,216
15824 :045,045,045,045,045,045,222
15830 :045,045,045,045,032,066,236
15836 :082,065,078,067,072,032,104
15842 :079,085,084,032,079,070,143
```

```
15848 :032,082,065,078,071,069,117
15854 :000,085,078,068,069,070,096
15860 :073,078,069,068,032,076,128
15866 :065,066,069,076,000,029,043
15872 :029,029,029,029,029,029,174
15878 :029,029,032,078,065,075,058
15884 :069,068,032,076,065,066,132
15890 :069,076,000,029,029,029,250
15896 :029,029,032,060,060,060,038
15902 :060,060,060,060,060,032,106
15908 :068,073,083,075,032,069,180
15914 :082,082,079,082,032,062,205
15920 :062,062,062,062,062,062,164
15926 :062,032,000,029,029,029,235
15932 :029,029,032,045,045,032,016
15938 :068,085,080,076,073,067,003
15944 :065,084,069,068,032,076,210
15950 :065,066,069,076,032,045,175
15956 :045,032,000,029,029,029,248
15962 :029,029,032,045,045,032,046
15968 :083,089,078,084,065,088,071
15974 :032,069,082,082,079,082,016
15980 :032,045,045,032,000,000,006
```

Program B-2. VIC Adjustments to Prog. B-1

To create the VIC-20 version of LADS, change the following lines in Program B-1:

```
11030 :141,157,062,185,000,016,071
11054 :061,200,185,000,016,201,197
12014 :255,076,116,196,185,089,131
12272 :152,032,205,221,173,139,138
12818 :133,188,032,190,225,096,114
12842 :133,188,032,190,225,032,074
12860 :000,133,183,032,190,225,055
12890 :188,032,114,225,032,204,117
13418 :008,232,189,158,192,016,133
13430 :192,048,007,153,068,061,135
14114 :008,232,189,158,192,016,061
14126 :192,048,007,153,000,002,192
14426 :096,169,000,032,205,221,045
14462 :032,205,221,032,022,057,183
14486 :221,032,076,057,096,169,033
14600 :140,062,032,205,221,032,188
14654 :166,253,032,205,221,032,203
14696 :221,032,204,255,162,001,211
```

Program B-3a. PET/CBM 4.0 BASIC Adjustments to Prog. B-1

To create the 4.0 BASIC version of LADS, type in Program B-1 then change the following bytes:

Address	Byte	Address	Byte	Address	Byte
2B05	BB	30F4	BD	324E	96
2B07	34	30F6	BE	3252	D4
2B0E	BC	30FA	BE	3256	DA
2B10	35	30FE	BD	325A	DB
2B1B	80	3012	BD	325C	56
2B32	80	3106	BE	325D	F3
2B39	D1	3108	BD	3262	28
2E07	C6	310E	BD	3264	BF
2E30	C6	3113	BE	3266	29
2E40	C6	3118	BD	3268	C0
2E47	C6	313C	BD	346D	B2
2EC0	E2	3143	BD	346E	B0
2EC1	F2	3148	BE	3475	B2
2ECE	E2	31A3	BD	3476	B0
2ECF	F2	31A9	BD	3496	A9
2ED3	E2	31DE	BD	3497	18
2ED4	F2	31E2	BE	3498	20
2EED	E2	31E4	BD	3499	D2
2EEE	F2	31FE	E2	349A	FF
2EF0	FF	31FF	F2	3725	B2
2EF1	B3	3203	D2	3726	B0
2FF2	83	3207	D4	372D	B2
2FF3	CF	320B	D3	372E	B0
3037	BB	320F	DA	385E	83
303C	BB	3213	DB	385F	CF
303E	BC	3215	63	387F	83
3042	BC	3216	F5	3800	CF
304B	BB	321B	D2	3895	83
3055	BB	321F	D4	3896	CF
3065	BB	3223	D3	390B	83
306A	BB	3227	DA	390C	CF
30C3	BB	322B	DB	3941	83
30C9	BB	322D	63	3942	CF
30D3	BD	322E	F5	3967	83
30D8	BE	3236	D2	3968	CF
30E3	BB	323A	D4	3A10	D1
30E5	BD	323E	D1	3ABE	D1
30E7	BC	3240	63	3B72	E2
30E9	BE	3241	F5	3B73	F2
30F0	BD	324C	9D	3BCE	96

Program B-3b. PET/CBM Upgrade BASIC Adjustments to Prog. B-1

To create the Upgrade BASIC version of LADS, type in Program B-1 then change the following bytes in *addition* to the changes shown in B-3a above:

Address	Byte	Address	Byte	Address	Byte
2EC0	AE	325C	22	387F	D9
2ECE	AE	346D	92	3880	DC
2ED3	AE	346E	C0	3895	D9
2EED	AE	3475	92	3896	DC
2EF0	89	3476	C0	390B	D9
2EF1	C3	3725	92	390C	DC
2FF2	D9	3726	C0	3941	D9
2FF3	DC	372D	92	3942	DC
31FE	AE	372E	C0	3967	D9
3215	24	385E	D9	3968	DC
322D	24	385F	DC	3B72	AE
3240	24				

Program B-4. Atari LADS: MLX Format

```
32768:076,203,146,169,000,133,215
32774:082,160,048,153,183,154,018
32780:136,208,250,169,000,133,140
32786:138,141,205,154,169,128,185
32792:133,139,141,206,154,169,198
32798:001,141,227,154,032,014,087
32804:145,165,162,208,026,160,134
32810:000,174,062,146,232,189,077
32816:000,005,201,155,240,008,145
32822:153,226,153,200,232,076,070
32828:047,128,132,128,032,013,028
32834:135,032,005,136,169,000,031
32840:141,189,154,032,190,136,146
32846:173,208,154,208,063,032,148
32852:121,141,169,160,032,036,231
32858:145,169,076,032,036,145,181
32864:169,065,032,036,145,169,200
32870:068,032,036,145,169,083,123
32876:032,036,145,032,121,141,103
32882:173,198,154,208,011,169,003
32888:144,133,134,169,153,133,218
32894:135,032,043,136,173,192,069
32900:154,133,136,141,185,154,011
32906:173,193,154,133,137,141,045
32912:186,154,032,175,145,173,241
```

```
32918:189,154,240,003,076,174,218
32924:131,032,190,136,169,000,046
32930:141,197,154,141,207,154,132
32936:172,208,154,208,003,076,221
32942:204,128,140,228,154,173,177
32948:226,154,240,012,032,130,206
32954:141,032,051,141,032,091,162
32960:141,032,051,141,173,219,181
32966:154,240,003,032,047,140,046
32972:076,183,135,173,184,154,085
32978:240,023,201,003,208,114,231
32984:169,001,141,184,154,173,014
32990:147,153,208,104,169,008,243
32996:024,109,183,154,141,183,254
33002:154,076,191,130,173,208,142
33008:154,240,057,160,255,200,026
33014:185,144,153,240,046,153,143
33020:226,153,201,032,208,243,035
33026:200,185,144,153,201,061,178
33032:208,003,076,239,130,162,058
33038:000,142,228,154,138,153,061
33044:226,153,185,144,153,240,097
33050:008,157,144,153,232,200,152
33056:076,022,129,157,144,153,201
33062:076,204,128,032,160,133,003
33068:032,066,133,076,204,128,171
33074:173,165,153,201,064,176,214
33080:006,173,166,153,238,207,231
33086:154,073,128,141,190,154,134
33092:032,228,133,076,203,129,101
33098:160,000,140,197,154,173,130
33104:147,153,201,032,240,003,088
33110:076,104,132,185,148,153,116
33116:201,065,144,003,238,197,172
33122:154,153,165,153,200,185,084
33128:148,153,240,022,153,165,217
33134:153,201,065,144,003,238,146
33140:197,154,200,185,148,153,129
33146:240,006,153,165,153,076,147
33152:118,129,136,140,196,154,233
33158:173,198,154,208,064,173,080
33164:197,154,208,162,169,165,171
33170:133,134,169,153,133,135,235
33176:160,000,173,165,153,201,236
33182:048,176,007,024,230,134,009
33188:144,002,230,135,177,134,218
33194:240,016,201,041,240,012,152
33200:201,044,240,008,201,032,134
33206:240,004,200,076,168,129,231
```

```
33212:072,152,072,169,000,145,030
33218:134,032,043,136,104,168,043
33224:104,145,134,173,165,153,050
33230:201,035,240,063,201,040,218
33236:240,023,173,184,154,201,163
33242:008,240,055,201,003,208,165
33248:113,169,008,024,109,183,062
33254:154,141,183,154,076,191,105
33260:130,172,196,154,185,165,214
33266:153,201,041,240,016,173,042
33272:184,154,201,001,208,009,237
33278:169,016,024,109,183,154,141
33284:141,183,154,173,184,154,225
33290:201,006,240,083,076,132,236
33296:130,076,159,130,173,208,124
33302:154,208,003,076,132,130,213
33308:056,173,192,154,229,136,200
33314:072,173,193,154,229,137,224
33320:176,014,201,255,240,004,162
33326:104,076,040,133,104,016,007
33332:012,076,068,130,240,004,070
33338:104,076,040,133,104,016,019
33344:003,076,040,133,056,233,093
33350:002,141,192,154,169,000,216
33356:141,193,154,076,132,130,134
33362:172,196,154,136,185,165,066
33368:153,201,044,208,004,200,130
33374:076,019,132,173,183,154,063
33380:201,076,208,003,076,141,037
33386:130,173,193,154,208,085,025
33392:173,184,154,201,006,176,238
33398:013,201,002,240,009,169,240
33404:004,024,109,183,154,141,227
33410:183,154,032,118,140,032,021
33416:156,140,076,239,130,172,025
33422:196,154,185,165,153,201,172
33428:041,208,005,169,108,141,052
33434:183,154,076,233,130,173,079
33440:166,153,201,034,208,006,160
33446:173,167,153,141,192,154,122
33452:173,184,154,201,001,208,069
33458:209,169,008,024,109,183,112
33464:154,141,183,154,076,132,000
33470:130,032,118,140,076,239,157
33476:130,173,184,154,201,002,016
33482:240,004,201,007,208,012,106
33488:173,183,154,024,105,008,087
33494:141,183,154,076,233,130,107
33500:201,006,176,009,173,183,200
```

starts Here

```
33506:154,024,105,012,141,183,077
33512:154,032,118,140,032,182,122
33518:140,173,208,154,208,003,100
33524:076,171,131,173,226,154,151
33530:208,003,076,171,131,173,244
33536:228,154,208,062,173,222,023
33542:154,240,042,169,020,056,175
33548:229,085,141,209,154,032,094
33554:014,145,162,004,032,011,130
33560:145,172,209,154,016,005,213
33566:160,002,076,037,131,169,093
33572:032,032,036,145,136,208,113
33578:250,032,014,145,162,001,134
33584:032,008,145,169,020,133,043
33590:085,169,226,133,134,169,202
33596:153,133,135,032,034,141,176
33602:169,030,056,229,085,141,008
33608:210,154,169,030,133,085,085
33614:173,222,154,240,031,032,162
33620:014,145,162,004,032,011,196
33626:145,172,210,154,240,010,253
33632:048,008,169,032,032,036,165
33638:145,136,208,250,032,014,119
33644:145,162,001,032,008,145,089
33650:032,143,141,173,220,154,209
33656:240,017,201,001,208,005,024
33662:169,060,076,133,131,169,096
33668:062,032,036,145,032,175,102
33674:141,173,229,154,240,019,070
33680:032,051,141,169,059,032,116
33686:036,145,169,000,133,134,255
33692:169,005,133,135,032,034,152
33698:141,032,121,141,173,189,191
33704:154,208,003,076,146,128,115
33710:173,208,154,208,041,238,172
33716:208,154,165,136,141,230,190
33722:154,165,137,141,231,154,144
33728:173,185,154,133,136,173,122
33734:186,154,133,137,032,014,086
33740:145,169,001,032,025,145,209
33746:165,162,208,003,032,013,025
33752:135,076,067,128,032,014,156
33758:145,169,001,032,025,145,227
33764:162,002,032,011,145,169,237
33770:000,032,036,145,032,014,237
33776:145,169,002,032,025,145,246
33782:173,222,154,240,021,032,064
33788:014,145,162,004,032,011,108
33794:145,169,013,032,036,145,030
```

```
33800:032,014,145,169,004,032,148
33806:025,145,076,182,145,185,004
33812:165,153,201,088,240,098,197
33818:136,136,185,165,153,201,234
33824:041,208,003,076,237,129,214
33830:173,193,154,208,015,173,186
33836:184,154,201,002,240,079,136
33842:201,005,240,075,201,001,005
33848:240,119,173,184,154,201,103
33854:001,208,012,173,183,154,025
33860:024,105,024,141,183,154,187
33866:076,233,130,173,184,154,000
33872:201,005,240,008,169,049,240
33878:032,248,132,076,104,132,042
33884:173,183,154,024,105,028,247
33890:141,183,154,076,233,130,247
33896:032,155,141,032,130,141,223
33902:169,157,133,134,169,154,002
33908:133,135,032,034,141,076,155
33914:239,130,173,193,154,208,195
33920:068,173,184,154,201,002,142
33926:208,012,169,016,024,109,160
33932:183,154,141,183,154,076,007
33938:132,130,201,001,240,016,098
33944:201,003,240,012,201,005,046
33950:240,008,169,050,032,248,137
33956:132,076,104,132,169,020,029
33962:024,109,183,154,141,183,196
33968:154,185,167,153,201,089,101
33974:208,010,173,183,154,201,087
33980:182,240,003,076,104,132,157
33986:076,132,130,173,184,154,019
33992:201,002,208,012,169,024,048
33998:024,109,183,154,141,183,232
34004:154,076,233,130,201,001,239
34010:240,016,201,003,240,012,162
34016:201,005,240,008,169,051,130
34022:032,248,132,076,104,132,186
34028:169,028,024,109,183,154,135
34034:141,183,154,076,233,130,135
34040:141,209,154,140,211,154,233
34046:142,210,154,169,160,032,097
34052:036,145,104,170,104,168,219
34058:152,072,138,072,152,032,116
34064:207,145,173,209,154,172,052
34070:211,154,174,210,154,096,253
34076:160,000,152,153,144,153,022
34082:200,192,080,208,248,096,034
34088:032,121,141,032,155,141,150
```

```
34094:032,130,141,169,018,133,157
34100:134,169,154,133,135,032,041
34106:034,141,032,121,141,076,091
34112:132,130,160,255,200,185,102
34118:144,153,240,086,201,032,158
34124:208,246,200,200,140,202,248
34130:154,056,165,138,237,202,010
34136:154,133,138,165,139,233,026
34142:000,133,139,160,000,185,199
34148:144,153,073,128,145,138,113
34154:200,185,144,153,201,032,253
34160:240,005,145,138,076,106,054
34166:133,200,185,144,153,201,110
34172:061,240,050,136,165,136,144
34178:145,138,200,165,137,145,036
34184:138,174,202,154,202,160,142
34190:000,189,144,153,240,008,108
34196:153,144,153,232,200,076,082
34202:143,133,153,144,153,096,208
34208:032,155,141,169,070,133,092
34214:134,169,154,133,135,032,155
34220:034,141,076,223,133,136,147
34226:140,203,154,173,198,154,176
34232:208,023,200,200,200,140,131
34238:191,154,169,144,024,109,213
34244:191,154,133,134,169,153,106
34250:105,000,133,135,032,043,138
34256:136,172,203,154,173,192,214
34262:154,145,138,173,193,154,147
34268:200,145,138,104,104,076,219
34274:239,130,173,205,154,133,236
34280:140,173,206,154,133,141,155
34286:032,242,134,169,255,141,187
34292:055,146,056,165,138,229,009
34298:140,165,139,229,141,176,216
34304:099,162,000,056,165,140,110
34310:233,002,133,140,165,141,052
34316:233,000,133,141,160,000,167
34322:177,140,048,012,165,140,188
34328:208,002,198,141,198,140,143
34334:232,076,018,134,165,140,027
34340:141,212,154,165,141,141,222
34346:213,154,177,140,205,190,097
34352:154,240,003,076,084,134,227
34358:232,142,191,154,162,001,168
34364:173,207,154,240,004,200,014
34370:032,242,134,200,185,165,000
34376:153,240,083,201,048,144,173
34382:079,232,209,140,240,241,195
```

```
34388:173,212,154,133,140,173,045
34394:213,154,133,141,032,242,237
34400:134,076,246,133,173,055,145
34406:146,048,001,096,173,208,006
34412:154,208,002,240,023,032,255
34418:155,141,032,130,141,032,233
34424:051,141,169,054,133,134,034
34430:169,154,133,135,032,034,015
34436:141,032,121,141,104,104,007
34442:173,183,154,041,031,201,153
34448:016,240,008,173,220,154,187
34454:208,003,076,233,130,076,108
34460:132,130,236,191,154,240,215
34466:003,076,084,134,238,055,240
34472:146,240,003,032,251,134,206
34478:172,191,154,173,207,154,201
34484:240,001,200,177,140,141,055
34490:192,154,200,177,140,141,166
34496:193,154,173,220,154,240,046
34502:010,201,002,208,030,173,054
34508:193,154,141,192,154,173,187
34514:219,154,240,019,024,173,015
34520:217,154,109,192,154,141,159
34526:192,154,173,218,154,109,198
34532:193,154,141,193,154,173,212
34538:208,154,240,001,096,076,241
34544:084,134,165,140,208,002,205
34550:198,141,198,140,096,032,027
34556:155,141,169,127,133,134,087
34562:169,154,133,135,032,034,147
34568:141,032,121,141,096,032,059
34574:014,145,169,001,032,025,144
34580:145,169,001,133,131,169,000
34586:004,133,133,169,000,133,086
34592:132,169,226,133,129,169,222
34598:153,133,130,032,218,144,080
34604:165,001,048,016,165,162,089
34610:240,011,032,003,152,169,145
34616:000,133,160,169,032,133,171
34622:161,096,032,115,150,076,180
34628:182,145,169,002,133,131,062
34634:169,008,133,133,169,000,174
34640:133,132,169,226,133,129,234
34646:169,153,133,130,169,002,074
34652:032,025,145,165,001,048,252
34658:221,032,218,144,162,002,109
34664:032,011,145,169,255,032,236
34670:036,145,032,036,145,173,165
34676:185,154,032,036,145,173,073
```

```
34682:186,154,032,036,145,173,080
34688:230,154,032,036,145,173,130
34694:231,154,032,036,145,032,252
34700:014,145,096,169,004,133,189
34706:131,032,025,145,169,008,144
34712:133,133,169,000,133,132,084
34718:169,002,133,128,169,181,172
34724:133,129,169,135,133,130,225
34730:032,218,144,165,001,048,010
34736:143,032,014,145,096,080,174
34742:058,160,000,162,255,232,025
34748:185,104,152,205,144,153,107
34754:240,010,200,200,200,224,244
34760:057,208,240,076,238,128,123
34766:200,185,104,152,205,145,173
34772:153,240,006,200,200,208,195
34778:224,240,238,200,185,104,129
34784:152,205,146,153,240,005,101
34790:200,208,210,240,224,173,205
34796:147,153,201,032,240,004,245
34802:201,000,208,213,189,016,045
34808:153,141,184,154,188,072,116
34814:153,140,183,154,076,207,143
34820:128,169,000,133,160,169,251
34826:032,133,161,162,001,032,019
34832:008,145,032,241,145,032,107
34838:085,145,201,042,240,014,237
34844:169,001,133,134,169,154,020
34850:133,135,032,034,141,076,073
34856:220,131,096,160,000,177,056
34862:134,201,048,144,008,201,014
34868:058,176,004,200,076,045,099
34874:136,140,254,153,136,169,022
34880:000,141,192,154,141,193,117
34886:154,162,001,142,210,154,125
34892:177,134,041,015,141,252,068
34898:153,141,255,153,169,000,185
34904:141,253,153,141,000,154,162
34910:202,240,018,032,131,136,085
34916:173,252,153,141,255,153,203
34922:173,253,153,141,000,154,212
34928:076,094,136,238,210,154,252
34934:174,210,154,032,170,136,226
34940:136,206,254,153,208,202,003
34946:096,024,014,252,153,046,203
34952:253,153,014,252,153,046,239
34958:253,153,024,173,255,153,129
34964:109,252,153,141,252,153,184
34970:173,000,154,109,253,153,228
```

```
34976:141,253,153,014,252,153,102
34982:046,253,153,096,024,173,143
34988:252,153,109,192,154,141,149
34994:192,154,173,253,153,109,188
35000:193,154,141,193,154,096,091
35006:032,028,133,160,000,140,171
35012:198,154,140,229,154,140,187
35018:220,154,140,219,154,173,238
35024:224,154,208,003,032,241,046
35030:145,032,085,145,208,008,069
35036:032,253,137,104,104,076,158
35042:146,128,201,032,240,239,188
35048:076,243,136,032,085,145,181
35054:208,003,076,253,137,201,092
35060:058,208,003,076,139,137,097
35066:201,059,208,104,140,209,147
35072:154,173,222,154,240,074,249
35078:141,229,154,173,209,154,042
35084:240,006,032,052,137,076,043
35090:088,137,032,085,145,240,233
35096:007,153,144,153,200,076,245
35102:020,137,032,130,141,032,010
35108:051,141,032,143,141,032,064
35114:121,141,169,000,141,209,055
35120:154,076,088,137,141,229,105
35126:154,141,209,154,160,000,104
35132:032,085,145,208,007,153,178
35138:000,005,172,209,154,096,190
35144:234,153,000,005,200,076,228
35150:060,137,032,085,145,240,009
35156:003,076,080,137,032,253,153
35162:137,173,209,154,208,005,208
35168:104,104,076,146,128,096,238
35174:201,062,240,047,201,060,145
35180:240,051,201,043,208,003,086
35186:238,219,154,201,042,208,152
35192:003,076,169,137,201,046,240
35198:240,015,201,036,240,014,104
35204:153,144,153,200,076,235,069
35210:136,141,224,154,096,076,197
35216:164,138,153,144,153,200,072
35222:076,031,138,169,002,141,195
35228:220,154,076,235,136,169,122
35234:001,141,220,154,076,235,221
35240:136,032,235,136,173,208,064
35246:154,240,011,169,042,032,054
35252:036,145,032,143,141,032,197
35258:121,141,173,198,154,208,157
35264:032,160,000,185,144,153,098
```

```
35270:201,032,240,004,200,076,183
35276:195,137,200,132,134,169,147
35282:144,024,101,134,133,134,112
35288:169,153,105,000,133,135,143
35294:032,043,136,173,208,154,200
35300:240,008,173,221,154,240,240
35306:003,032,238,139,173,192,243
35312:154,133,136,173,193,154,159
35318:133,137,104,104,076,146,178
35324:128,153,144,153,200,192,198
35330:080,208,248,153,144,153,220
35336:173,083,003,201,003,240,199
35342:010,201,136,240,006,169,008
35348:000,141,224,154,096,169,036
35354:001,141,189,154,096,162,001
35360:000,032,085,145,240,044,066
35366:201,058,240,040,201,032,042
35372:240,243,201,059,240,032,035
35378:201,044,240,015,201,041,024
35384:240,011,157,205,153,232,030
35390:153,144,153,200,076,033,053
35396:138,142,199,154,153,144,230
35402:153,200,032,102,138,076,007
35408:235,136,141,209,154,169,100
35414:000,142,199,154,153,144,110
35420:153,032,102,138,173,209,131
35426:154,076,238,136,169,000,103
35432:141,192,154,141,193,154,055
35438:170,014,192,154,046,193,111
35444:154,014,192,154,046,193,101
35450:154,014,192,154,046,193,107
35456:154,014,192,154,046,193,113
35462:154,189,205,153,201,065,077
35468:144,002,233,007,041,015,070
35474:013,192,154,141,192,154,224
35480:232,236,199,154,208,209,110
35486:238,198,154,169,001,096,246
35492:192,000,240,014,174,208,224
35498:154,208,009,072,152,072,069
35504:032,066,133,104,168,104,015
35510:153,144,153,200,032,085,181
35516:145,153,144,153,200,201,160
35522:066,208,104,169,000,141,114
35528:214,154,173,208,154,240,063
35534:023,140,211,154,173,226,109
35540:154,240,015,032,130,141,156
35546:032,051,141,032,091,141,194
35552:032,051,141,172,211,154,217
35558:032,085,145,153,144,153,174
```

```
35564:200,201,032,208,245,032,130
35570:085,145,153,144,153,200,098
35576:201,034,208,069,032,085,109
35582:145,208,003,076,211,139,012
35588:201,058,208,003,076,214,252
35594:139,201,059,208,012,032,149
35600:052,137,174,222,154,142,129
35606:229,154,076,211,139,201,008
35612:034,208,003,076,252,138,227
35618:174,208,154,208,009,032,051
35624:020,141,076,252,138,076,231
35630:122,142,153,144,153,170,162
35636:140,211,154,032,236,140,197
35642:172,211,154,200,076,252,099
35648:138,162,000,142,215,154,107
35654:157,245,153,232,173,215,221
35660:154,208,117,032,085,145,049
35666:240,067,201,058,240,063,183
35672:201,059,208,012,032,052,140
35678:137,174,222,154,142,229,128
35684:154,076,151,139,141,185,178
35690:153,173,208,154,208,013,247
35696:173,185,153,201,032,208,040
35702:211,032,020,141,076,074,160
35708:139,173,185,153,153,144,047
35714:153,200,201,032,240,024,212
35720:201,000,240,020,201,058,088
35726:240,016,157,245,153,232,161
35732:076,074,139,238,215,154,020
35738:141,186,153,076,104,139,185
35744:169,245,133,134,169,153,139
35750:133,135,140,211,154,032,203
35756:043,136,174,192,154,032,135
35762:236,140,172,211,154,169,236
35768:000,162,005,157,245,153,138
35774:202,208,250,076,074,139,115
35780:173,208,154,208,003,032,206
35786:020,141,173,186,153,201,052
35792:058,240,003,032,253,137,163
35798:141,224,154,238,228,154,073
35804:104,104,173,208,154,240,179
35810:008,173,226,154,240,003,006
35816:076,114,131,076,146,128,135
35822:173,208,154,201,002,208,160
35828:001,096,032,014,145,162,182
35834:002,032,011,145,056,173,157
35840:192,154,229,136,141,190,018
35846:154,173,193,154,229,137,022
35852:141,191,154,169,000,032,187
```

```
35858:036,145,173,190,154,208,156
35864:003,206,191,154,206,190,206
35870:154,208,238,173,191,154,124
35876:208,233,032,014,145,162,062
35882:001,032,008,145,096,160,228
35888:000,162,000,185,144,153,180
35894:201,043,240,004,200,076,050
35900:051,140,200,185,144,153,165
35906:032,078,140,176,018,157,155
35912:205,153,232,076,062,140,172
35918:201,058,176,006,056,233,040
35924:048,056,233,208,096,169,126
35930:000,157,205,153,169,205,211
35936:133,134,169,153,133,135,185
35942:032,043,136,173,192,154,064
35948:141,217,154,173,193,154,116
35954:141,218,154,096,173,208,080
35960:154,208,004,032,020,141,167
35966:096,173,226,154,240,017,008
35972:032,014,145,162,001,032,006
35978:008,145,174,183,154,032,066
35984:060,141,032,051,141,174,231
35990:183,154,032,236,140,096,223
35996:173,208,154,208,004,032,167
36002:020,141,096,173,226,154,204
36008:240,006,174,192,154,032,198
36014:060,141,174,192,154,076,203
36020:236,140,173,208,154,208,019
36026:007,032,020,141,032,020,182
36032:141,096,173,226,154,240,198
36038:006,174,192,154,032,060,048
36044:141,174,192,154,032,236,109
36050:140,173,226,154,240,014,133
36056:173,227,154,240,003,032,021
36062:051,141,174,193,154,032,199
36068:060,141,174,193,154,076,002
36074:236,140,142,191,154,173,246
36080:223,154,240,005,160,000,254
36086:138,145,136,173,221,154,189
36092:240,022,032,014,145,162,099
36098:002,032,011,145,173,191,044
36104:154,032,042,145,032,014,171
36110:145,162,001,032,008,145,251
36116:024,169,001,101,136,133,072
36122:136,169,000,101,137,133,190
36128:137,096,160,000,177,134,224
36134:240,010,032,036,145,032,021
36140:169,141,200,076,036,141,039
36146:096,169,032,032,036,145,048
```

```
36152:032,169,141,096,142,210,078
36158:154,173,227,154,240,011,253
36164:138,032,097,142,032,210,207
36170:141,174,210,154,096,169,250
36176:000,032,207,145,032,210,194
36182:141,174,210,154,096,173,010
36188:227,154,240,014,165,137,005
36194:032,097,142,165,136,032,190
36200:097,142,032,005,142,096,106
36206:166,136,165,137,032,207,185
36212:145,032,005,142,096,169,193
36218:013,032,036,145,032,169,037
36224:141,096,174,187,154,173,029
36230:188,154,032,207,145,032,124
36236:059,142,096,169,144,133,115
36242:134,169,153,133,135,032,134
36248:034,141,096,169,253,032,109
36254:036,145,032,143,141,169,056
36260:013,032,036,145,096,174,148
36266:208,154,208,001,096,174,243
36272:222,154,208,001,096,141,230
36278:209,154,032,014,145,162,130
36284:004,032,011,145,173,209,250
36290:154,032,036,145,032,014,095
36296:145,162,001,032,008,145,181
36302:173,209,154,096,174,208,196
36308:154,208,001,096,174,222,043
36314:154,208,001,096,032,014,211
36320:145,162,004,032,011,145,211
36326:173,227,154,240,009,173,182
36332:210,154,032,097,142,076,179
36338:252,141,169,000,174,210,164
36344:154,032,207,145,032,014,064
36350:145,162,001,032,008,145,235
36356:096,174,208,154,208,001,077
36362:096,174,222,154,208,001,097
36368:096,032,014,145,162,004,213
36374:032,011,145,174,227,154,253
36380:240,013,165,137,032,097,200
36386:142,165,136,032,097,142,236
36392:076,050,142,165,137,166,008
36398:136,032,207,145,032,014,100
36404:145,162,001,032,008,145,033
36410:096,174,208,154,208,001,131
36416:096,174,222,154,208,001,151
36422:096,032,014,145,162,004,011
36428:032,011,145,173,188,154,011
36434:174,187,154,032,207,145,213
36440:032,014,145,162,001,032,218
```

```
36446:008,145,096,072,041,015,215
36452:168,185,128,153,170,104,240
36458:074,074,074,074,168,185,243
36464:128,153,032,036,145,138,232
36470:032,036,145,096,201,070,186
36476:208,008,032,221,142,104,071
36482:104,076,146,128,201,069,086
36488:208,006,032,039,143,076,128
36494:129,142,201,068,208,003,125
36500:076,102,143,201,080,208,190
36506:003,076,171,143,201,078,058
36512:208,003,076,236,143,201,003
36518:079,208,003,076,215,143,122
36524:201,083,208,003,076,165,140
36530:144,201,072,208,003,076,114
36536:191,144,153,144,153,032,233
36542:130,141,032,051,141,032,205
36548:091,141,032,155,141,032,020
36554:143,141,169,157,133,134,055
36560:169,154,133,135,032,034,097
36566:141,032,121,141,076,190,147
36572:143,032,085,145,201,032,090
36578:240,003,076,221,142,160,044
36584:000,032,085,145,201,000,183
36590:240,007,153,144,153,200,111
36596:076,233,142,132,128,160,091
36602:000,185,144,153,240,008,212
36608:153,226,153,200,196,128,032
36614:208,243,173,208,154,208,176
36620:006,032,091,141,032,051,109
36626:141,032,143,141,032,121,116
36632:141,032,013,135,162,001,252
36638:032,008,145,162,000,142,007
36644:189,154,096,169,046,032,210
36650:036,145,169,069,032,036,017
36656:145,169,078,032,036,145,141
36662:169,068,032,036,145,169,161
36668:032,032,036,145,032,221,046
36674:142,173,208,154,240,003,218
36680:238,189,154,238,208,154,229
36686:165,136,141,230,154,165,045
36692:137,141,231,154,173,185,081
36698:154,133,136,173,186,154,002
36704:133,137,032,190,136,096,052
36710:173,208,154,240,023,032,164
36716:085,145,153,144,153,160,180
36722:000,032,085,145,240,013,117
36728:153,144,153,153,226,153,078
36734:200,076,115,143,076,190,158
```

```
36740:143,132,128,032,143,141,083
36746:032,121,141,238,221,154,021
36752:032,070,135,032,014,145,060
36758:162,001,032,008,145,032,018
36764:132,144,032,253,137,104,190
36770:104,162,000,142,189,154,145
36776:076,146,128,173,208,154,029
36782:240,014,032,143,135,238,208
36788:222,154,032,014,145,162,141
36794:001,032,008,145,032,085,233
36800:145,240,007,201,058,240,059
36806:006,076,190,143,032,253,130
36812:137,104,104,162,000,142,085
36818:189,154,076,146,128,169,048
36824:046,032,036,145,169,079,211
36830:032,036,145,032,121,141,217
36836:169,001,141,223,154,076,224
36842:190,143,173,208,154,240,062
36848:205,032,085,145,201,080,220
36854:240,012,201,079,240,058,052
36860:201,083,240,106,201,072,131
36866:240,076,169,046,032,036,089
36872:145,169,078,032,036,145,101
36878:169,080,032,036,145,032,252
36884:121,141,206,222,154,032,128
36890:014,145,162,004,032,011,138
36896:145,169,013,032,036,145,060
36902:169,004,032,025,145,032,189
36908:014,145,162,001,032,008,150
36914:145,076,190,143,169,046,051
36920:032,036,145,169,078,032,036
36926:036,145,169,079,032,036,047
36932:145,032,121,141,169,000,164
36938:141,223,154,076,190,143,233
36944:169,046,032,036,145,169,165
36950:078,032,036,145,169,072,106
36956:032,036,145,032,121,141,087
36962:169,000,141,227,154,076,097
36968:190,143,169,046,032,036,208
36974:145,169,078,032,036,145,203
36980:169,083,032,036,145,032,101
36986:121,141,169,000,141,226,152
36992:154,076,190,143,174,099,196
36998:003,048,001,096,169,000,195
37004:032,060,141,032,051,141,085
37010:169,092,133,134,169,154,229
37016:133,135,032,155,141,032,012
37022:034,141,104,104,076,220,069
37028:131,169,046,032,036,145,211
```

```
37034:169,083,032,036,145,032,155
37040:121,141,173,208,154,240,189
37046:005,169,001,141,226,154,110
37052:076,190,143,169,046,032,076
37058:036,145,169,072,032,036,172
37064:145,032,121,141,169,001,041
37070:141,227,154,076,190,143,113
37076:010,010,010,010,170,096,006
37082:165,131,032,212,144,165,043
37088:129,157,068,003,165,130,108
37094:157,069,003,165,128,157,141
37100:072,003,169,000,157,073,198
37106:003,165,133,157,074,003,009
37112:165,132,157,075,003,169,181
37118:003,157,066,003,032,086,089
37124:228,132,001,096,134,142,225
37130:096,134,143,096,162,000,129
37136:134,142,134,143,134,131,066
37142:134,001,096,032,212,144,129
37148:169,012,157,066,003,076,255
37154:002,145,201,013,208,002,093
37160:169,155,141,203,145,140,225
37166:204,145,142,205,145,165,028
37172:143,032,212,144,169,000,240
37178:157,072,003,157,073,003,011
37184:169,011,157,066,003,173,131
37190:203,145,032,002,145,172,001
37196:204,145,174,205,145,173,098
37202:203,145,096,140,204,145,247
37208:142,205,145,165,162,240,123
37214:046,160,000,177,160,072,197
37220:230,160,208,002,230,161,067
37226:024,165,160,237,047,146,117
37232:141,206,145,165,161,237,143
37238:048,146,013,206,145,144,052
37244:005,240,003,076,174,131,241
37250:169,000,133,001,141,083,145
37256:003,104,076,162,145,165,023
37262:142,032,212,144,169,000,073
37268:157,072,003,157,073,003,101
37274:169,007,157,066,003,032,076
37280:002,145,172,204,145,174,234
37286:205,145,201,155,208,002,058
37292:169,000,096,072,165,017,179
37298:240,002,104,096,076,203,131
37304:146,162,007,142,206,145,224
37310:138,032,025,145,174,206,142
37316:145,202,208,243,076,014,060
37322:145,000,000,000,000,134,225
```

```
37328:212,133,213,032,170,217,161
37334:032,230,216,160,000,140,224
37340:240,145,177,243,072,041,114
37346:127,032,036,145,104,048,206
37352:006,172,240,145,200,208,179
37358:236,096,000,160,000,032,250
37364:085,145,201,032,240,007,186
37370:153,000,005,200,076,243,159
37376:145,169,000,153,000,005,216
37382:169,000,133,134,169,005,104
37388:133,135,032,043,136,173,152
37394:192,154,141,187,154,173,251
37400:193,154,141,188,154,160,246
37406:000,096,076,203,146,000,039
37412:000,000,000,000,000,000,036
37418:000,000,000,000,000,000,042
37424:000,000,000,000,000,000,048
37430:000,000,000,000,000,000,054
37436:000,000,000,000,000,000,060
37442:000,173,035,146,141,104,153
37448:146,173,036,146,141,105,051
37454:146,173,037,146,141,107,060
37460:146,173,038,146,141,108,068
37466:146,174,040,146,240,032,100
37472:169,000,141,041,146,160,241
37478:000,185,255,255,153,255,181
37484:255,200,204,041,146,208,138
37490:244,238,105,146,238,108,169
37496:146,224,000,240,008,202,172
37502:208,224,173,039,146,208,100
37508:221,096,173,040,146,170,210
37514:013,039,146,208,001,096,129
37520:024,138,109,036,146,141,226
37526:184,146,173,035,146,141,207
37532:183,146,024,138,109,038,026
37538:146,141,187,146,173,037,224
37544:146,141,186,146,232,172,167
37550:039,146,208,004,240,013,056
37556:160,255,185,255,255,153,163
37562:255,255,136,192,255,208,207
37568:245,206,184,146,206,187,086
37574:146,202,208,234,096,162,222
37580:255,154,032,185,145,169,120
37586:000,133,162,169,002,133,041
37592:082,032,121,141,169,240,233
37598:141,124,148,169,150,141,071
37604:125,148,169,228,141,126,141
37610:148,169,148,141,127,148,091
37616:169,084,141,128,148,169,055
```

```
37622:147,141,129,148,169,175,131
37628:141,130,148,169,151,141,108
37634:131,148,169,246,141,132,201
37640:148,169,151,141,133,148,130
37646:169,196,141,134,148,169,203
37652:151,141,135,148,169,211,207
37658:141,136,148,169,151,141,144
37664:137,148,169,224,141,138,221
37670:148,169,151,141,139,148,166
37676:169,074,141,006,002,169,093
37682:152,141,007,002,173,065,078
37688:146,240,003,076,087,147,243
37694:169,203,141,065,146,076,094
37700:084,147,169,000,141,047,144
37706:146,169,032,141,048,146,244
37712:032,014,145,096,032,070,213
37718:147,169,142,160,150,032,118
37724:146,149,160,000,140,063,238
37730:146,140,064,146,032,085,199
37736:145,166,001,016,017,224,161
37742:136,240,007,224,128,240,061
37748:003,032,115,150,032,104,040
37754:150,076,087,147,201,034,049
37760:208,010,072,173,064,146,033
37766:073,001,141,064,146,104,151
37772:201,048,208,005,174,063,071
37778:146,240,209,238,063,146,164
37784:201,059,208,003,238,064,157
37790:146,174,064,146,208,012,140
37796:041,127,201,097,144,006,012
37802:201,123,176,002,041,095,040
37808:153,000,005,200,201,000,223
37814:208,174,136,169,155,153,153
37820:000,005,140,042,146,192,201
37826:000,240,153,173,000,005,253
37832:201,058,176,039,201,048,155
37838:176,003,076,243,147,169,252
37844:255,032,223,150,165,208,221
37850:141,045,146,173,055,146,156
37856:208,003,032,163,149,172,183
37862:045,146,204,042,146,240,029
37868:003,032,255,149,076,094,077
37874:147,169,082,133,203,169,121
37880:148,133,204,160,000,140,009
37886:046,146,162,000,177,203,220
37892:240,048,201,255,240,034,254
37898:221,000,005,208,009,232,173
37904:200,208,239,230,204,076,149
37910:002,148,177,203,240,008,032
```

```
37916:200,208,249,230,204,076,171
37922:024,148,238,046,146,162,030
37928:000,076,016,148,169,156,093
37934:160,150,032,146,149,076,247
37940:087,147,142,062,146,173,041
37946:046,146,010,170,189,124,231
37952:148,056,233,001,141,066,197
37958:146,189,125,148,233,000,143
37964:072,173,066,146,072,096,189
37970:076,073,083,084,000,068,210
37976:079,083,000,078,069,087,228
37982:000,083,065,086,069,032,173
37988:000,076,079,065,068,032,164
37994:000,077,069,082,071,069,218
38000:032,000,076,065,068,083,180
38006:000,083,089,083,000,255,116
38012:000,000,000,000,000,000,124
38018:000,000,000,000,000,000,130
38024:000,000,000,000,169,000,049
38030:133,203,056,173,047,146,132
38036:229,203,141,039,146,169,051
38042:032,133,204,173,048,146,122
38048:229,204,141,040,146,173,069
38054:040,146,170,013,039,146,208
38060:208,001,096,169,001,141,020
38066:254,002,224,000,240,029,159
38072:169,000,141,049,146,160,081
38078:000,177,203,032,036,145,015
38084:165,001,048,022,200,204,068
38090:049,146,208,241,230,204,000
38096:202,048,011,208,234,173,060
38102:039,146,141,049,146,076,043
38108:189,148,169,000,141,254,097
38114:002,096,108,010,000,169,099
38120:000,133,203,169,032,133,134
38126:204,169,000,141,055,146,185
38132:168,140,050,146,152,024,156
38138:101,203,133,134,141,051,245
38144:146,141,053,146,165,204,087
38150:105,000,133,135,141,052,060
38156:146,141,054,146,056,173,216
38162:051,146,237,047,146,141,018
38168:066,146,173,052,146,237,076
38174:048,146,013,066,146,144,081
38180:003,076,097,149,032,043,180
38186:136,056,173,192,154,237,222
38192:043,146,141,066,146,173,251
38198:193,154,237,044,146,013,073
38204:066,146,240,013,176,014,203
```

```
38210:032,128,149,200,208,002,017
38216:230,204,076,245,148,206,157
38222:055,146,032,128,149,024,100
38228:152,101,203,141,053,146,112
38234:169,000,101,204,141,054,247
38240:146,238,055,146,056,173,142
38246:053,146,237,051,146,141,108
38252:056,146,173,054,146,237,152
38258:052,146,141,057,146,238,126
38264:056,146,208,003,238,057,060
38270:146,096,172,050,146,177,145
38276:203,201,155,240,008,200,115
38282:208,247,230,204,076,131,210
38288:149,096,133,203,132,204,037
38294:160,000,177,203,240,006,168
38300:032,036,145,200,208,246,255
38306:096,173,053,146,024,105,247
38312:001,141,035,146,173,054,206
38318:146,105,000,141,036,146,236
38324:173,051,146,141,037,146,106
38330:173,052,146,141,038,146,114
38336:056,173,047,146,237,053,136
38342:146,141,039,146,173,048,123
38348:146,237,054,146,176,014,209
38354:173,047,146,240,003,206,001
38360:048,146,206,047,146,076,117
38366:235,149,141,040,146,013,178
38372:039,146,240,022,032,067,006
38378:146,056,173,047,146,237,015
38384:056,146,141,047,146,173,181
38390:048,146,237,057,146,141,253
38396:048,146,096,173,051,146,144
38402:133,203,141,035,146,056,204
38408:109,042,146,141,037,146,117
38414:173,052,146,133,204,141,095
38420:036,146,105,000,141,038,230
38426:146,056,173,047,146,237,063
38432:051,146,141,039,146,173,216
38438:048,146,237,052,146,141,040
38444:040,146,176,014,173,047,128
38450:146,208,003,206,048,146,039
38456:206,047,146,076,070,150,239
38462:013,039,146,240,003,032,023
38468:134,146,056,173,047,146,002
38474:109,042,146,141,047,146,193
38480:173,048,146,105,000,141,181
38486:048,146,160,000,185,000,113
38492:005,145,203,200,204,042,123
38498:146,144,245,240,243,096,188
```

```
38504:165,131,240,003,032,025,188
38510:145,032,014,145,096,165,195
38516:001,141,066,146,032,185,175
38522:145,169,171,160,150,032,181
38528:146,149,174,066,146,169,210
38534:000,032,207,145,032,121,159
38540:141,096,155,076,065,068,229
38546:083,032,082,101,097,100,129
38552:121,046,155,000,253,083,042
38558:121,110,116,097,120,032,242
38564:069,114,114,111,114,155,073
38570:000,253,069,114,114,111,063
38576:114,032,045,032,000,066,209
38582:082,075,032,102,114,111,186
38588:109,032,000,133,242,230,166
38594:242,169,000,133,243,169,126
38600:005,133,244,032,000,216,062
38606:176,008,032,210,217,165,246
38612:242,133,208,096,169,000,036
38618:133,212,133,213,096,032,013
38624:191,150,165,212,141,043,102
38630:146,165,213,141,044,146,061
38636:032,231,148,096,032,246,253
38642:150,076,087,147,173,062,169
38648:146,205,042,146,208,003,230
38654:076,140,148,032,223,150,255
38660:173,051,146,141,060,146,209
38666:173,052,146,141,061,146,217
38672:173,053,146,141,058,146,221
38678:173,054,146,141,059,146,229
38684:165,208,205,042,146,208,234
38690:020,173,055,146,208,047,171
38696:173,058,146,141,053,146,245
38702:173,059,146,141,054,146,253
38708:076,058,151,032,223,150,230
38714:173,060,146,133,203,173,178
38720:061,146,133,204,056,173,069
38726:053,146,229,203,141,039,113
38732:146,173,054,146,229,204,004
38738:141,040,146,176,001,096,170
38744:173,055,146,208,008,238,148
38750:039,146,208,003,238,040,000
38756:146,076,165,148,024,173,064
38762:062,146,105,000,133,129,169
38768:169,000,105,005,133,130,142
38774:172,062,146,185,000,005,176
38780:201,155,240,010,201,044,207
38786:240,006,200,208,242,076,078
38792:044,148,152,056,237,062,067
```

```
38798:146,140,062,146,133,128,129
38804:169,007,133,131,032,025,133
38810:145,169,000,133,132,032,253
38816:218,144,166,001,048,001,226
38822:096,104,104,032,115,150,255
38828:076,087,147,169,008,133,024
38834:133,032,104,151,166,131,127
38840:032,011,145,032,246,150,032
38846:032,104,150,076,087,147,018
38852:169,004,133,133,032,104,003
38858:151,166,131,032,008,145,067
38864:076,094,147,173,062,146,138
38870:205,042,146,208,002,230,023
38876:162,076,003,128,173,062,056
38882:146,032,191,150,165,212,098
38888:141,241,151,165,213,141,004
38894:242,151,032,255,255,076,225
38900:087,147,032,252,151,076,221
38906:087,147,169,004,133,133,155
38912:032,104,151,165,131,032,103
38918:212,144,169,000,157,068,244
38924:003,169,032,157,069,003,189
38930:169,000,157,072,003,169,076
38936:080,157,073,003,169,007,001
38942:157,066,003,032,002,145,179
38948:165,131,032,212,144,024,232
38954:189,072,003,105,000,141,040
38960:047,146,189,073,003,105,099
38966:032,141,048,146,165,001,075
38972:201,136,240,006,032,115,022
38978:150,076,087,147,032,104,150
38984:150,096,088,169,181,160,148
38990:150,032,146,149,104,104,251
38996:104,056,233,002,170,104,241
39002:233,000,032,207,145,162,101
39008:255,154,032,121,141,076,107
39014:203,146,076,068,065,076,224
39020:068,089,074,083,082,082,074
39026:084,083,066,067,083,066,051
39032:069,081,066,067,067,067,025
39038:077,080,066,078,069,076,060
39044:068,088,074,077,080,083,090
39050:084,065,083,084,089,083,114
39056:084,088,073,078,089,068,112
39062:069,089,068,069,088,068,089
39068:069,067,073,078,088,073,092
39074:078,067,067,080,089,067,098
39080:080,088,083,066,067,083,123
39086:069,067,065,068,067,067,065
```

```
39092:076,067,084,065,088,084,132
39098:065,089,084,088,065,084,149
39104:089,065,080,072,065,080,131
39110:076,065,066,082,075,066,116
39116:077,073,066,080,076,065,129
39122:078,068,079,082,065,069,139
39128:079,082,066,073,084,066,154
39134:086,067,066,086,083,082,180
39140:079,076,082,079,082,076,190
39146:083,082,067,076,068,067,165
39152:076,073,065,083,076,080,181
39158:072,080,080,076,080,082,204
39164:084,073,083,069,068,083,200
39170:069,073,084,083,088,084,227
39176:088,083,067,076,086,078,230
39182:079,080,001,005,009,000,188
39188:008,008,008,001,008,005,058
39194:006,001,002,002,000,000,037
39200:000,002,000,002,004,004,044
39206:001,000,001,000,000,000,040
39212:000,000,000,000,000,008,052
39218:008,001,001,001,007,008,076
39224:008,003,003,003,000,000,073
39230:003,000,000,000,000,000,065
39236:000,000,000,000,161,160,133
39242:032,096,176,240,144,193,187
39248:208,162,076,129,132,134,153
39254:200,136,202,198,232,230,004
39260:192,224,225,056,097,024,142
39266:170,168,138,152,072,104,134
39272:000,048,016,033,001,065,011
39278:036,080,112,034,098,066,024
39284:216,088,002,008,040,064,022
39290:248,120,186,154,184,234,224
39296:048,049,050,051,052,053,175
39302:054,055,056,057,065,066,231
39308:067,068,069,070,000,000,158
39314:000,000,000,000,000,000,146
39320:000,000,000,000,000,000,152
39326:000,000,000,000,000,000,158
39332:000,000,000,000,000,000,164
39338:000,000,000,000,000,000,170
39344:000,000,000,000,000,000,176
39350:000,000,000,000,000,000,182
39356:000,000,000,000,000,000,188
39362:000,000,000,000,000,000,194
39368:000,000,000,000,000,000,200
39374:000,000,000,000,000,000,206
39380:000,000,000,000,000,000,212
```

```
39386:000,000,000,000,000,000,218
39392:000,000,000,000,000,000,224
39398:000,000,000,000,000,000,230
39404:000,000,000,000,000,000,236
39410:000,000,000,000,000,000,242
39416:000,000,000,000,000,000,248
39422:000,000,000,206,239,160,091
39428:211,244,225,242,244,160,050
39434:193,228,228,242,229,243,093
39440:243,000,045,045,045,045,183
39446:045,045,045,045,045,045,036
39452:045,045,045,045,045,032,029
39458:194,242,225,238,227,232,112
39464:160,207,245,244,160,239,015
39470:230,160,210,225,238,231,060
39476:229,000,213,238,228,229,165
39482:230,233,238,229,228,160,096
39488:204,225,226,229,236,000,160
39494:031,031,031,031,031,031,000
39500:031,031,031,032,206,225,120
39506:235,229,228,160,236,225,115
39512:226,229,236,000,031,031,073
39518:031,031,031,032,188,188,083
39524:188,188,188,188,188,188,204
39530:160,196,201,211,203,160,213
39536:197,210,210,207,210,160,026
39542:190,190,190,190,190,190,234
39548:190,190,000,031,031,031,085
39554:031,031,160,173,173,160,090
39560:196,245,240,236,233,227,233
39566:225,244,229,160,160,204,084
39572:225,226,229,236,160,173,117
39578:173,160,000,031,031,031,068
39584:031,031,160,173,173,160,120
39590:211,249,238,244,225,248,045
39596:160,197,242,242,239,242,214
39602:160,173,173,160,000,000,076
```

Program B-5. Apple LADS: Hex DATA

```
79FD- 4C F5 82
7A00- A9 00 A0 32 99 CE 8F 88
7A08- D0 FA A9 00 85 EB 85 4C
7A10- 8D E4 8F A9 7A 85 EC 85
7A18- 4D 8D E5 8F A9 01 8D FA
7A20- 8F B9 00 04 C9 A0 F0 07
7A28- 99 F3 8E C8 4C 21 7A 99
7A30- F3 8E C8 B9 00 04 C9 A0
7A38- D0 E7 88 84 F9 20 E5 80
7A40- 20 58 83 A9 00 8D D4 8F
```

```
7A48- 20 0E 84 AD E7 8F D0 3F
7A50- 20 50 89 A9 E6 20 D6 81
7A58- A9 4C 20 D6 81 A9 41 20
7A60- D6 81 A9 44 20 D6 81 A9
7A68- 53 20 D6 81 20 50 89 AD
7A70- DD 8F D0 0B A9 F1 85 FB
7A78- A9 8D 85 FC 20 81 83 AD
7A80- D7 8F 85 FD 8D D0 8F AD
7A88- D8 8F 85 FE 8D D1 8F 20
7A90- 2F 82 AD D4 8F F0 03 4C
7A98- A1 7D 20 0E 84 A9 00 8D
7AA0- DC 8F 8D E6 8F AC E7 8F
7AA8- D0 03 4C C9 7A 8C FB 8F
7AB0- AD F9 8F F0 0C 20 59 89
7AB8- 20 0A 89 20 32 89 20 0A
7AC0- 89 AD F2 8F F0 03 20 06
7AC8- 88 4C 0A 83 AD CF 8F F0
7AD0- 17 C9 03 D0 72 A9 01 8D
7AD8- CF 8F AD F4 8D D0 68 A9
7AE0- 08 18 6D CE 8F 8D CE 8F
7AE8- 4C B2 7C AD E7 8F F0 39
7AF0- A0 FF C8 B9 F1 8D F0 2E
7AF8- 99 F3 8E C9 20 D0 F3 C8
7B00- B9 F1 8D C9 3D D0 03 4C
7B08- E2 7C A2 00 8E FB 8F 8A
7B10- 99 F3 8E B9 F1 8D F0 08
7B18- 9D F1 8D E8 C8 4C 13 7B
7B20- 9D F1 8D 4C C9 7A 20 78
7B28- 7F 20 1A 7F 4C C9 7A AD
7B30- 38 8E C9 40 B0 06 AD 39
7B38- 8E EE E6 8F 49 80 8D D5
7B40- 8F 20 BC 7F 4C BE 7B A0
7B48- 00 8C DC 8F B9 F5 8D C9
7B50- 41 90 03 EE DC 8F 99 38
7B58- 8E C8 B9 F5 8D F0 16 99
7B60- 38 8E C9 41 90 03 EE DC
7B68- 8F C8 B9 F5 8D F0 06 99
7B70- 38 8E 4C 69 7B 88 8C DB
7B78- 8F AD DD 8F D0 40 AD DC
7B80- 8F D0 AC A9 38 85 FB A9
7B88- 8E 85 FC A0 00 AD 38 8E
7B90- C9 30 B0 07 18 E6 FB 90
7B98- 02 E6 FC B1 FB F0 10 C9
7BA0- 29 F0 0C C9 2C F0 08 C9
7BA8- 20 F0 04 C8 4C 9B 7B 48
7BB0- 98 48 A9 00 91 FB 20 81
7BB8- 83 68 A8 68 91 FB AD 38
7BC0- 8E C9 23 F0 3F C9 28 F0
7BC8- 17 AD CF 8F C9 08 F0 37
```

```
7BD0- C9 03 D0 71 A9 08 18 6D
7BD8- CE 8F 8D CE 8F 4C B2 7C
7BE0- AC DB 8F B9 38 8E C9 29
7BE8- F0 10 AD CF 8F C9 01 D0
7BF0- 09 A9 10 18 6D CE 8F 8D
7BF8- CE 8F AD CF 8F C9 06 F0
7C00- 53 4C 77 7C 4C 92 7C AD
7C08- E7 8F D0 03 4C 77 7C 38
7C10- AD D7 8F E5 FD 48 AD D8
7C18- 8F E5 FE B0 0E C9 FF F0
7C20- 04 68 4C 00 7F 68 10 0C
7C28- 4C 37 7C F0 04 68 4C 00
7C30- 7F 68 10 03 4C 00 7F 38
7C38- E9 02 8D D7 8F A9 00 8D
7C40- D8 8F 4C 77 7C AC DB 8F
7C48- 88 B9 38 8E C9 2C D0 04
7C50- C8 4C FC 7D AD CE 8F C9
7C58- 4C D0 03 4C 80 7C AD D8
7C60- 8F D0 55 AD CF 8F C9 06
7C68- B0 0D C9 02 F0 09 A9 04
7C70- 18 6D CE 8F 8D CE 8F 20
7C78- 4D 88 20 73 88 4C E2 7C
7C80- AC DB 8F B9 38 8E C9 29
7C88- D0 05 A9 6C 8D CE 8F 4C
7C90- DC 7C AD 39 8E C9 22 D0
7C98- 06 AD 3A 8E 8D D7 8F AD
7CA0- CF 8F C9 01 D0 D1 A9 08
7CA8- 18 6D CE 8F 8D CE 8F 4C
7CB0- 77 7C 20 4D 88 4C E2 7C
7CB8- AD CF 8F C9 02 F0 04 C9
7CC0- 07 D0 0C AD CE 8F 18 69
7CC8- 08 8D CE 8F 4C DC 7C C9
7CD0- 06 B0 09 AD CE 8F 18 69
7CD8- 0C 8D CE 8F 20 4D 88 20
7CE0- 8D 88 AD E7 8F D0 03 4C
7CE8- 9E 7D AD F9 8F D0 03 4C
7CF0- 9E 7D AD FB 8F D0 3E AD
7CF8- F5 8F F0 2A A9 14 38 E5
7D00- 24 8D E8 8F 20 1C 82 A2
7D08- 04 20 A6 81 AC E8 8F 10
7D10- 05 A0 02 4C 18 7D A9 20
7D18- 20 D6 81 88 D0 FA 20 1C
7D20- 82 A2 01 20 A2 81 A9 14
7D28- 85 24 A9 F3 85 FB A9 8E
7D30- 85 FC 20 F9 88 A9 1E 38
7D38- E5 24 8D E9 8F A9 1E 85
7D40- 24 AD F5 8F F0 1F 20 1C
7D48- 82 A2 04 20 A6 81 AC E9
7D50- 8F F0 0A 30 08 A9 20 20
```

```
7D58- D6 81 88 D0 FA 20 1C 82
7D60- A2 01 20 A2 81 20 66 89
7D68- AD F3 8F F0 11 C9 01 D0
7D70- 05 A9 3C 4C 78 7D A9 3E
7D78- 20 D6 81 20 8B 89 AD FC
7D80- 8F F0 13 20 0A 89 A9 3B
7D88- 20 D6 81 A9 00 85 FB A9
7D90- 02 85 FC 20 F9 88 20 50
7D98- 89 AD D4 8F D0 03 4C 8F
7DA0- 7A AD E7 8F D0 2C EE E7
7DA8- 8F 38 A5 FD ED D0 8F 8D
7DB0- FD 8F A5 FE ED D1 8F 8D
7DB8- FE 8F AD D0 8F 85 FD AD
7DC0- D1 8F 85 FE 20 1C 82 A9
7DC8- 01 20 35 82 20 E5 80 4C
7DD0- 40 7A 20 1C 82 A9 01 20
7DD8- 35 82 A9 02 20 35 82 AD
7DE0- F5 8F F0 15 20 1C 82 A2
7DE8- 04 20 A6 81 A9 0D 20 D6
7DF0- 81 20 1C 82 A9 04 20 35
7DF8- 82 4C D0 03 B9 38 8E C9
7E00- 58 F0 62 88 88 B9 38 8E
7E08- C9 29 D0 03 4C E0 7B AD
7E10- D8 8F D0 0F AD CF 8F C9
7E18- 02 F0 4F C9 05 F0 4B C9
7E20- 01 F0 77 AD CF 8F C9 01
7E28- D0 0C AD CE 8F 18 69 18
7E30- 8D CE 8F 4C DC 7C AD CF
7E38- 8F C9 05 F0 08 A9 31 20
7E40- D0 7E 4C 51 7E AD CE 8F
7E48- 18 69 1C 8D CE 8F 4C DC
7E50- 7C 20 72 89 20 59 89 A9
7E58- B4 85 FB A9 8F 85 FC 20
7E60- F9 88 4C E2 7C AD D8 8F
7E68- D0 33 AD CF 8F C9 02 D0
7E70- 0C A9 10 18 6D CE 8F 8D
7E78- CE 8F 4C 77 7C C9 01 F0
7E80- 10 C9 03 F0 0C C9 05 F0
7E88- 08 A9 32 20 D0 7E 4C 51
7E90- 7E A9 14 18 6D CE 8F 8D
7E98- CE 8F 4C 77 7C AD CF 8F
7EA0- C9 02 D0 0C A9 18 18 6D
7EA8- CE 8F 8D CE 8F 4C DC 7C
7EB0- C9 01 F0 10 C9 03 F0 0C
7EB8- C9 05 F0 08 A9 33 20 D0
7EC0- 7E 4C 51 7E A9 1C 18 6D
7EC8- CE 8F 8D CE 8F 4C DC 7C
7ED0- 8D E8 8F 8C EA 8F 8E E9
7ED8- 8F A9 BA 20 D6 81 68 AA
```

```
7EE0- 68 A8 98 48 8A 48 98 20
7EE8- 24 ED AD E8 8F AC EA 8F
7EF0- AE E9 8F 60 A0 00 98 99
7EF8- F1 8D C8 C0 FF D0 F8 60
7F00- 20 50 89 20 72 89 20 59
7F08- 89 A9 23 85 FB A9 8F 85
7F10- FC 20 F9 88 20 50 89 4C
7F18- 77 7C A0 FF C8 B9 F1 8D
7F20- F0 56 C9 20 D0 F6 C8 C8
7F28- 8C E1 8F 38 A5 EB ED E1
7F30- 8F 85 EB A5 EC E9 00 85
7F38- EC A0 00 B9 F1 8D 49 80
7F40- 91 EB C8 B9 F1 8D C9 20
7F48- F0 05 91 EB 4C 42 7F C8
7F50- B9 F1 8D C9 3D F0 32 88
7F58- A5 FD 91 EB C8 A5 FE 91
7F60- EB AE E1 8F CA A0 00 BD
7F68- F1 8D F0 08 99 F1 8D E8
7F70- C8 4C 67 7F 99 F1 8D 60
7F78- 20 72 89 A9 5C 85 FB A9
7F80- 8F 85 FC 20 F9 88 4C B7
7F88- 7F 88 8C E2 8F AD DD 8F
7F90- D0 17 C8 C8 C8 8C D6 8F
7F98- A9 F1 18 6D D6 8F 85 FB
7FA0- A9 8D 69 00 85 FC 20 81
7FA8- 83 AC E2 8F AD D7 8F 91
7FB0- EB AD D8 8F C8 91 EB 68
7FB8- 68 4C E2 7C AD E4 8F 85
7FC0- ED AD E5 8F 85 EE 20 CA
7FC8- 80 A9 FF 8D F8 8F 38 A5
7FD0- EB E5 ED A5 EC E5 EE B0
7FD8- 63 A2 00 38 A5 ED E9 02
7FE0- 85 ED A5 EE E9 00 85 EE
7FE8- A0 00 B1 ED 30 0C A5 ED
7FF0- D0 02 C6 EE C6 ED E8 4C
7FF8- EA 7F A5 ED 8D EB 8F A5
8000- EE 8D EC 8F B1 ED CD D5
8008- 8F F0 03 4C 2C 80 E8 8E
8010- D6 8F A2 01 AD E6 8F F0
8018- 04 C8 20 CA 80 C8 B9 38
8020- 8E F0 53 C9 30 90 4F E8
8028- D1 ED F0 F1 AD EB 8F 85
8030- ED AD EC 8F 85 EE 20 CA
8038- 80 4C CE 7F AD F8 8F 30
8040- 01 60 AD E7 8F D0 02 F0
8048- 17 20 72 89 20 59 89 20
8050- 0A 89 A9 4C 85 FB A9 8F
8058- 85 FC 20 F9 88 20 50 89
8060- 68 68 AD CE 8F 29 1F C9
```

```
8068- 10 F0 08 AD F3 8F D0 03
8070- 4C DC 7C 4C 77 7C EC D6
8078- 8F F0 03 4C 2C 80 EE F8
8080- 8F F0 03 20 D3 80 AC D6
8088- 8F AD E6 8F F0 01 C8 B1
8090- ED 8D D7 8F C8 B1 ED 8D
8098- D8 8F AD F3 8F F0 0A C9
80A0- 02 D0 1E AD D8 8F 8D D7
80A8- 8F AD F2 8F F0 13 18 AD
80B0- F0 8F 6D D7 8F 8D D7 8F
80B8- AD F1 8F 6D D8 8F 8D D8
80C0- 8F AD E7 8F F0 01 60 4C
80C8- 2C 80 A5 ED D0 02 C6 EE
80D0- C6 ED 60 20 72 89 A9 96
80D8- 85 FB A9 8F 85 FC 20 F9
80E0- 88 20 50 89 60 20 1C 82
80E8- A9 01 20 35 82 A9 01 85
80F0- 2C A9 90 85 2D 20 5F 81
80F8- EE FF 8F 60 A9 13 85 2C
8100- A9 90 85 2D 20 5F 81 EE
8108- 00 90 60 60 A9 25 85 2C
8110- A9 90 85 2D 20 8A 81 20
8118- DC 03 85 2B 84 2A A0 08
8120- B1 2A 60 8D 3F 90 A9 37
8128- 85 2C A9 90 85 2D 20 8A
8130- 81 60 AD FF 8F F0 27 A9
8138- 49 85 2C A9 90 85 2D 20
8140- 8A 81 A9 00 8D FF 8F 60
8148- AD 00 90 F0 11 A9 5B 85
8150- 2C A9 90 85 2D 20 8A 81
8158- A9 00 8D 00 90 60 60 A0
8160- 08 B1 2C 85 2A C8 B1 2C
8168- 85 2B A9 F3 85 FB A9 8E
8170- 85 FC A0 00 A9 A0 91 2A
8178- C8 C0 1F D0 F9 A0 00 B1
8180- FB 09 80 91 2A C8 C4 F9
8188- D0 F5 20 DC 03 85 2B 84
8190- 2A A0 00 B1 2C 91 2A C8
8198- C0 12 D0 F7 A2 00 20 D6
81A0- 03 60 8E 6D 90 60 8A 8D
81A8- 6E 90 E0 04 D0 0A A9 EC
81B0- 8D 53 AA A9 81 8D 54 AA
81B8- 60 8C 70 90 8E E9 8F AD
81C0- 6D 90 C9 01 D0 0C 20 0C
81C8- 81 08 AC 70 90 AE E9 8F
81D0- 28 60 AC 70 90 60 8C 70
81D8- 90 8D 6F 90 AD 6E 90 C9
81E0- 02 D0 1E AD 6F 90 20 23
81E8- 81 4C D2 81 8D 6F 90 C9
```

```
81F0- 8D D0 02 A9 0A 8D 90 C0
81F8- AD C1 C1 30 FB AD 6F 90
8200- 60 AD 6E 90 C9 04 D0 09
8208- AD 6F 90 20 EC 81 4C D2
8210- 81 AD 6F 90 09 80 20 F0
8218- FD 4C D2 81 A9 00 8D 6E
8220- 90 8D 6D 90 A9 F0 8D 53
8228- AA A9 FD 8D 54 AA 60 AD
8230- 00 C0 C9 83 60 C9 01 D0
8238- 03 4C 32 81 C9 02 D0 03
8240- 4C 48 81 4C 5E 81 8D 6F
8248- 90 A9 00 C5 B8 D0 1B A9
8250- 02 C5 B9 D0 15 A0 00 B1
8258- B8 C9 20 D0 05 E6 B8 4C
8260- 57 82 C9 2F 90 04 C9 3A
8268- 90 53 AD 00 02 C9 41 D0
8270- 37 AD 01 02 C9 53 D0 30
8278- AD 02 02 C9 4D D0 29 AD
8280- 03 02 C9 20 D0 22 A0 00
8288- B9 04 02 C9 00 F0 09 09
8290- 80 99 00 04 C8 4C 88 82
8298- A9 A0 99 00 04 99 01 04
82A0- 99 02 04 68 68 4C 00 7A
82A8- AD 6F 90 C9 3A B0 0D C9
82B0- 20 D0 03 4C B1 00 38 E9
82B8- 30 38 E9 D0 60 A6 AF 86
82C0- 69 A6 B0 86 6A 18 20 0C
82C8- DA 20 D1 82 68 68 4C 6A
82D0- D4 A0 00 84 94 A9 02 85
82D8- 95 B1 B8 91 94 C8 C9 00
82E0- D0 F7 88 88 B1 94 C9 20
82E8- F0 F9 C8 A9 00 91 94 C8
82F0- C8 C8 C8 C8 60 A9 46 85
82F8- BB A9 82 85 BC A9 4C 85
8300- BA A9 FC 85 73 A9 79 85
8308- 74 60 A0 00 A2 FF E8 B9
8310- C9 8C CD F1 8D F0 0A C8
8318- C8 C8 E0 39 D0 F0 4C EB
8320- 7A C8 B9 C9 8C CD F2 8D
8328- F0 06 C8 C8 D0 E0 F0 EE
8330- C8 B9 C9 8C CD F3 8D F0
8338- 05 C8 D0 D2 F0 E0 AD F4
8340- 8D C9 20 F0 04 C9 00 D0
8348- D5 BD 71 8D 8D CF 8F BC
8350- A9 8D 8C CE 8F 4C CC 7A
8358- A2 01 20 A2 81 A2 06 8E
8360- E9 8F 20 B9 81 AE E9 8F
8368- CA D0 F4 20 B9 81 C9 2A
8370- F0 0E A9 12 85 FB A9 8F
```

```
8378- 85 FC 20 F9 88 4C D2 7D
8380- 60 A0 00 B1 FB F0 04 C8
8388- 4C 83 83 8C 0F 8F 88 A9
8390- 00 8D D7 8F 8D D8 8F A2
8398- 01 8E E9 8F B1 FB 29 0F
83A0- 8D 0D 8F 8D 10 8F A9 00
83A8- 8D 0E 8F 8D 11 8F CA F0
83B0- 12 20 D3 83 AD 0D 8F 8D
83B8- 10 8F AD 0E 8F 8D 11 8F
83C0- 4C AE 83 EE E9 8F AE E9
83C8- 8F 20 FA 83 88 CE 0F 8F
83D0- D0 CA 60 18 0E 0D 8F 2E
83D8- 0E 8F 0E 0D 8F 2E 0E 8F
83E0- 18 AD 10 8F 6D 0D 8F 8D
83E8- 0D 8F AD 11 8F 6D 0E 8F
83F0- 8D 0E 8F 0E 0D 8F 2E 0E
83F8- 8F 60 18 AD 0D 8F 6D D7
8400- 8F 8D D7 8F AD 0E 8F 6D
8408- D8 8F 8D D8 8F 60 20 F4
8410- 7E A0 00 8C DD 8F 8C FC
8418- 8F 8C F3 8F 8C F2 8F AD
8420- F7 8F D0 0C 20 B9 81 8D
8428- D2 8F 20 B9 81 8D D3 8F
8430- 20 B9 81 C9 20 D0 08 20
8438- B2 85 68 68 4C 8F 7A C9
8440- 20 4C 4C 84 20 B9 81 D0
8448- 03 4C B2 85 C9 3A D0 03
8450- 4C F6 84 C9 3B D0 73 8C
8458- E8 8F AD F5 8F F0 55 8D
8460- FC 8F AD E8 8F F0 06 20
8468- 94 84 4C BC 84 20 B9 81
8470- F0 0E C9 7F 90 03 20 04
8478- 85 99 F1 8D C8 4C 6D 84
8480- 20 59 89 20 0A 89 20 66
8488- 89 20 50 89 A9 00 8D E8
8490- 8F 4C BC 84 8D FC 8F 8D
8498- E8 8F A0 00 20 B9 81 D0
84A0- 07 99 00 02 AC E8 8F 60
84A8- 10 03 20 E1 87 99 00 02
84B0- C8 4C 9C 84 20 B9 81 F0
84B8- 03 4C B4 84 20 B2 85 AD
84C0- E8 8F D0 05 68 68 4C 8F
84C8- 7A 60 C9 3E F0 5B C9 3C
84D0- F0 5F C9 2B D0 03 EE F2
84D8- 8F C9 2A D0 03 4C 39 85
84E0- C9 2E F0 16 C9 24 F0 15
84E8- C9 7F 90 03 20 04 85 99
84F0- F1 8D C8 4C 44 84 8D F7
84F8- 8F 60 4C 56 86 99 F1 8D
```

```
8500- C8 4C D1 85 38 E9 7F 8D
8508- E0 8F A2 FF CE E0 8F F0
8510- 08 E8 BD D0 D0 10 FA 30
8518- F3 E8 BD D0 D0 30 07 99
8520- F1 8D C8 4C 19 85 29 7F
8528- 60 A9 02 8D F3 8F 4C 44
8530- 84 A9 01 8D F3 8F 4C 44
8538- 84 AD F3 8F F0 20 A9 2A
8540- 99 F1 8D C8 EE DD 8F AD
8548- F3 8F C9 01 F0 08 A5 FE
8550- 8D D7 8F 4C 44 84 A5 FD
8558- 8D D7 8F 4C 44 84 20 44
8560- 84 AD E7 8F F0 0B A9 2A
8568- 20 D6 81 20 66 89 20 50
8570- 89 AD DD 8F D0 20 A0 00
8578- B9 F1 8D C9 20 F0 04 C8
8580- 4C 78 85 C8 84 FB A9 F1
8588- 18 65 FB 85 FB A9 8D 69
8590- 00 85 FC 20 81 83 AD E7
8598- 8F F0 08 AD F4 8F F0 03
85A0- 20 A0 87 AD D7 8F 85 FD
85A8- AD D8 8F 85 FE 68 68 4C
85B0- 8F 7A 99 F1 8D C8 C0 FF
85B8- D0 F8 99 F1 8D 20 B9 81
85C0- 20 B9 81 F0 06 A9 00 8D
85C8- F7 8F 60 A9 01 8D D4 8F
85D0- 60 A2 00 20 B9 81 F0 2C
85D8- C9 3A F0 28 C9 20 F0 F3
85E0- C9 3B F0 20 C9 2C F0 0F
85E8- C9 29 F0 0B 9D DE 8E E8
85F0- 99 F1 8D C8 4C D3 85 8E
85F8- DE 8F 99 F1 8D C8 20 18
8600- 86 4C 44 84 8D E8 8F A9
8608- 00 8E DE 8F 99 F1 8D 20
8610- 18 86 AD E8 8F 4C 47 84
8618- A9 00 8D D7 8F 8D D8 8F
8620- AA 0E D7 8F 2E D8 8F 0E
8628- D7 8F 2E D8 8F 0E D7 8F
8630- 2E D8 8F 0E D7 8F 2E D8
8638- 8F BD DE 8E C9 41 90 02
8640- E9 07 29 0F 0D D7 8F 8D
8648- D7 8F E8 EC DE 8F D0 D1
8650- EE DD 8F A9 01 60 C0 00
8658- F0 0E AE E7 8F D0 09 48
8660- 98 48 20 1A 7F 68 A8 68
8668- 99 F1 8D C8 20 B9 81 99
8670- F1 8D C8 C9 42 D0 68 A9
8678- 00 8D ED 8F AD E7 8F F0
8680- 17 8C EA 8F AD F9 8F F0
```

```
8688- 0F 20 59 89 20 0A 89 20
8690- 32 89 20 0A 89 AC EA 8F
8698- 20 B9 81 99 F1 8D C8 C9
86A0- 20 D0 F5 20 B9 81 99 F1
86A8- 8D C8 C9 22 D0 45 20 B9
86B0- 81 D0 03 4C 85 87 C9 3A
86B8- D0 03 4C 88 87 C9 3B D0
86C0- 0C 20 94 84 AE F5 8F 8E
86C8- FC 8F 4C 85 87 C9 22 D0
86D0- 03 4C AE 86 AE E7 8F D0
86D8- 09 20 EB 88 4C AE 86 4C
86E0- 56 8A 99 F1 8D AA 8C EA
86E8- 8F 20 C3 88 AC EA 8F C8
86F0- 4C AE 86 A2 00 8E EE 8F
86F8- 9D 06 8F E8 AD EE 8F D0
8700- 75 20 B9 81 F0 43 C9 3A
8708- F0 3F C9 3B D0 0C 20 94
8710- 84 AE F5 8F 8E FC 8F 4C
8718- 49 87 8D 80 8E AD E7 8F
8720- D0 0D AD 80 8E C9 20 D0
8728- D3 20 EB 88 4C FC 86 AD
8730- 80 8E 99 F1 8D C8 C9 20
8738- F0 18 C9 00 F0 14 C9 3A
8740- F0 10 9D 06 8F E8 4C FC
8748- 86 EE EE 8F 8D 81 8E 4C
8750- 1A 87 A9 06 85 FB A9 8F
8758- 85 FC 8C EA 8F 20 81 83
8760- AE D7 8F 20 C3 88 AC EA
8768- 8F A9 00 A2 05 9D 06 8F
8770- CA D0 FA 4C FC 86 AD E7
8778- 8F D0 03 20 EB 88 AD 81
8780- 8E C9 3A F0 03 20 B2 85
8788- 8D F7 8F EE FB 8F 68 68
8790- AD E7 8F F0 08 AD F9 8F
8798- F0 03 4C 65 7D 4C 8F 7A
87A0- AD E7 8F C9 02 D0 01 60
87A8- 20 1C 82 A2 02 20 A6 81
87B0- 38 AD D7 8F E5 FD 8D D5
87B8- 8F AD D8 8F E5 FE 8D D6
87C0- 8F A9 00 20 D6 81 AD D5
87C8- 8F D0 03 CE D6 8F CE D5
87D0- 8F D0 EE AD D6 8F D0 E9
87D8- 20 1C 82 A2 01 20 A2 81
87E0- 60 38 E9 7F 8D E0 8F A2
87E8- FF CE E0 8F F0 08 E8 BD
87F0- D0 D0 10 FA 30 F3 E8 BD
87F8- D0 D0 30 07 99 00 02 C8
8800- 4C F6 87 29 7F 60 A0 00
8808- A2 00 B9 F1 8D C9 2B F0
```

```
8810- 04 C8 4C 0A 88 C8 B9 F1
8818- 8D 20 25 88 B0 12 9D DE
8820- 8E E8 4C 15 88 C9 3A B0
8828- 06 38 E9 30 38 E9 D0 60
8830- A9 00 9D DE 8E A9 DE 85
8838- FB A9 8E 85 FC 20 81 83
8840- AD D7 8F 8D F0 8F AD D8
8848- 8F 8D F1 8F 60 AD E7 8F
8850- D0 04 20 EB 88 60 AD F9
8858- 8F F0 11 20 1C 82 A2 01
8860- 20 A2 81 AE CE 8F 20 13
8868- 89 20 0A 89 AE CE 8F 20
8870- C3 88 60 AD E7 8F D0 04
8878- 20 EB 88 60 AD F9 8F F0
8880- 06 AE D7 8F 20 13 89 AE
8888- D7 8F 4C C3 88 AD E7 8F
8890- D0 07 20 EB 88 20 EB 88
8898- 60 AD F9 8F F0 06 AE D7
88A0- 8F 20 13 89 AE D7 8F 20
88A8- C3 88 AD F9 8F F0 0E AD
88B0- FA 8F F0 03 20 0A 89 AE
88B8- D8 8F 20 13 89 AE D8 8F
88C0- 4C C3 88 8E D6 8F AD F6
88C8- 8F F0 05 A0 00 8A 91 FD
88D0- AD F4 8F F0 16 20 1C 82
88D8- A2 02 20 A6 81 AD D6 8F
88E0- 20 D6 81 20 1C 82 A2 01
88E8- 20 A2 81 18 A9 01 65 FD
88F0- 85 FD A9 00 65 FE 85 FE
88F8- 60 A0 00 B1 FB F0 0A 20
8900- D6 81 20 85 89 C8 4C FB
8908- 88 60 A9 20 20 D6 81 20
8910- 85 89 60 8E E9 8F AD FA
8918- 8F F0 0B 8A 20 3D 8A 20
8920- AE 89 AE E9 8F 60 A9 00
8928- 20 24 ED 20 AE 89 AE E9
8930- 8F 60 AD FA 8F F0 0E A5
8938- FE 20 3D 8A A5 FD 20 3D
8940- 8A 20 E1 89 60 A6 FD A5
8948- FE 20 24 ED 20 E1 89 60
8950- A9 0D 20 D6 81 20 85 89
8958- 60 AE D2 8F AD D3 8F 20
8960- 24 ED 20 17 8A 60 A9 F1
8968- 85 FB A9 8D 85 FC 20 F9
8970- 88 60 A9 07 20 D6 81 A9
8978- 12 20 D6 81 20 66 89 A9
8980- 0D 20 D6 81 60 AE E7 8F
8988- D0 01 60 AE F5 8F D0 01
8990- 60 8D E8 8F 20 1C 82 A2
```

```
8998- 04 20 A6 81 AD E8 8F 20
89A0- D6 81 20 1C 82 A2 01 20
89A8- A2 81 AD E8 8F 60 AE E7
89B0- 8F D0 01 60 AE F5 8F D0
89B8- 01 60 20 1C 82 A2 04 20
89C0- A6 81 AD FA 8F F0 09 AD
89C8- E9 8F 20 3D 8A 4C D8 89
89D0- A9 00 AE E9 8F 20 24 ED
89D8- 20 1C 82 A2 01 20 A2 81
89E0- 60 AE E7 8F D0 01 60 AE
89E8- F5 8F D0 01 60 20 1C 82
89F0- A2 04 20 A6 81 AE FA 8F
89F8- F0 0D A5 FE 20 3D 8A A5
8A00- FD 20 3D 8A 4C 0E 8A A5
8A08- FE A6 FD 20 24 ED 20 1C
8A10- 82 A2 01 20 A2 81 60 AE
8A18- E7 8F D0 01 60 AE F5 8F
8A20- D0 01 60 20 1C 82 A2 04
8A28- 20 A6 81 AD D3 8F AE D2
8A30- 8F 20 24 ED 20 1C 82 A2
8A38- 01 20 A2 81 60 48 29 0F
8A40- A8 B9 E1 8D AA 68 4A 4A
8A48- 4A 4A A8 B9 E1 8D 20 D6
8A50- 81 8A 20 D6 81 60 C9 46
8A58- D0 08 20 B9 8A 68 68 4C
8A60- 8F 7A C9 45 D0 06 20 12
8A68- 8B 4C 5D 8A C9 44 D0 03
8A70- 4C 5B 8B C9 50 D0 03 4C
8A78- C1 8B C9 4E D0 03 4C 02
8A80- 8C C9 4F D0 03 4C ED 8B
8A88- C9 53 D0 03 4C 9A 8C C9
8A90- 48 D0 03 4C B4 8C 99 F1
8A98- 8D 20 59 89 20 0A 89 20
8AA0- 32 89 20 72 89 20 66 89
8AA8- A9 B4 85 FB A9 8F 85 FC
8AB0- 20 F9 88 20 50 89 4C D4
8AB8- 8B 20 B9 81 C9 20 F0 03
8AC0- 4C B9 8A A0 00 20 B9 81
8AC8- C9 00 F0 0E C9 7F 90 03
8AD0- 20 04 85 99 F1 8D C8 4C
8AD8- C5 8A 84 F9 A0 00 B9 F1
8AE0- 8D F0 07 99 F3 8E C8 4C
8AE8- DE 8A AD E7 8F D0 06 20
8AF0- 32 89 20 0A 89 20 66 89
8AF8- 20 50 89 20 E5 80 A2 01
8B00- 20 A2 81 20 B9 81 20 B9
8B08- 81 20 B2 85 A2 00 8E D4
8B10- 8F 60 A9 2E 20 D6 81 A9
8B18- 45 20 D6 81 A9 4E 20 D6
```

```
8B20- 81 A9 44 20 D6 81 A9 20
8B28- 20 D6 81 20 B9 81 20 B9
8B30- 8A AD E7 8F F0 03 EE D4
8B38- 8F EE E7 8F 38 A5 FD ED
8B40- D0 8F 8D FD 8F A5 FE ED
8B48- D1 8F 8D FE 8F AD D0 8F
8B50- 85 FD AD D1 8F 85 FE 20
8B58- 0E 84 60 AD E7 8F F0 1E
8B60- 20 B9 81 99 F1 8D A0 00
8B68- 20 B9 81 F0 14 C9 7F 90
8B70- 03 20 04 85 99 F1 8D 99
8B78- F3 8E C8 4C 68 8B 4C D4
8B80- 8B 84 F9 20 66 89 20 50
8B88- 89 EE F4 8F 20 FC 80 A2
8B90- 02 20 A6 81 AD D0 8F 20
8B98- D6 81 AD D1 8F 20 D6 81
8BA0- AD FD 8F 20 D6 81 AD FE
8BA8- 8F 20 D6 81 20 1C 82 A2
8BB0- 01 20 A2 81 20 B2 85 68
8BB8- 68 A2 00 8E D4 8F 4C 8F
8BC0- 7A AD E7 8F F0 0E 20 0B
8BC8- 81 EE F5 8F 20 1C 82 A2
8BD0- 01 20 A2 81 20 B9 81 F0
8BD8- 07 C9 3A F0 06 4C D4 8B
8BE0- 20 B2 85 68 68 A2 00 8E
8BE8- D4 8F 4C 8F 7A A9 2E 20
8BF0- D6 81 A9 4F 20 D6 81 20
8BF8- 50 89 A9 01 8D F6 8F 4C
8C00- D4 8B AD E7 8F F0 CD 20
8C08- B9 81 C9 50 F0 0C C9 4F
8C10- F0 3A C9 53 F0 6A C9 48
8C18- F0 4C A9 2E 20 D6 81 A9
8C20- 4E 20 D6 81 A9 50 20 D6
8C28- 81 20 50 89 CE F5 8F 20
8C30- 1C 82 A2 04 20 A6 81 A9
8C38- 0D 20 D6 81 A9 04 20 35
8C40- 82 20 1C 82 A2 01 20 A2
8C48- 81 4C D4 8B A9 2E 20 D6
8C50- 81 A9 4E 20 D6 81 A9 4F
8C58- 20 D6 81 20 50 89 A9 00
8C60- 8D F6 8F 4C D4 8B A9 2E
8C68- 20 D6 81 A9 4E 20 D6 81
8C70- A9 48 20 D6 81 20 50 89
8C78- A9 00 8D FA 8F 4C D4 8B
8C80- A9 2E 20 D6 81 A9 4E 20
8C88- D6 81 A9 53 20 D6 81 20
8C90- 50 89 A9 00 8D F9 8F 4C
8C98- D4 8B A9 2E 20 D6 81 A9
8CA0- 53 20 D6 81 20 50 89 AD
```

```
8CA8- E7 8F F0 05 A9 01 8D F9
8CB0- 8F 4C D4 8B A9 2E 20 D6
8CB8- 81 A9 48 20 D6 81 20 50
8CC0- 89 A9 01 8D FA 8F 4C D4
8CC8- 8B 4C 44 41 4C 44 59 4A
8CD0- 53 52 52 54 53 42 43 53
8CD8- 42 45 51 42 43 43 43 4D
8CE0- 50 42 4E 45 4C 44 58 4A
8CE8- 4D 50 53 54 41 53 54 59
8CF0- 53 54 58 49 4E 59 44 45
8CF8- 59 44 45 58 44 45 43 49
8D00- 4E 58 49 4E 43 43 50 59
8D08- 43 50 58 53 42 43 53 45
8D10- 43 41 44 43 43 4C 43 54
8D18- 41 58 54 41 59 54 58 41
8D20- 54 59 41 50 48 41 50 4C
8D28- 41 42 52 4B 42 4D 49 42
8D30- 50 4C 41 4E 44 4F 52 41
8D38- 45 4F 52 42 49 54 42 56
8D40- 43 42 56 53 52 4F 4C 52
8D48- 4F 52 4C 53 52 43 4C 44
8D50- 43 4C 49 41 53 4C 50 48
8D58- 50 50 4C 50 52 54 49 53
8D60- 45 44 53 45 49 54 53 58
8D68- 54 58 53 43 4C 56 4E 4F
8D70- 50 01 05 09 00 08 08 08
8D78- 01 08 05 06 01 02 02 00
8D80- 00 00 02 00 02 04 04 01
8D88- 00 01 00 00 00 00 00 00
8D90- 00 00 08 08 01 01 01 07
8D98- 08 08 03 03 03 00 00 03
8DA0- 00 00 00 00 00 00 00 00
8DA8- 00 A1 A0 20 60 B0 F0 90
8DB0- C1 D0 A2 4C 81 84 86 C8
8DB8- 88 CA C6 E8 E6 C0 E0 E1
8DC0- 38 61 18 AA A8 BA 98 48
8DC8- 68 00 30 10 21 01 41 24
8DD0- 50 70 22 62 42 D8 58 02
8DD8- 08 28 40 F8 78 BA 9A B8
8DE0- EA 30 31 32 33 34 35 36
8DE8- 37 38 39 41 42 43 44 45
8DF0- 46 00 00 00 00 00 00 00
8DF8- 00 00 00 00 00 00 00 00
8E00- 00 00 00 00 00 00 00 00
8E08- 00 00 00 00 00 00 00 00
8E10- 00 00 00 00 00 00 00 00
8E18- 00 00 00 00 00 00 00 00
8E20- 00 00 00 00 00 00 00 00
8E28- 00 00 00 00 00 00 00 00
```

```
8E30- 00 00 00 00 00 00 00 00
8E38- 00 00 00 00 00 00 00 00
8E40- 00 00 00 00 00 00 00 00
8E48- 00 00 00 00 00 00 00 00
8E50- 00 00 00 00 00 00 00 00
8E58- 00 00 00 00 00 00 00 00
8E60- 00 00 00 00 00 00 00 00
8E68- 00 00 00 00 00 00 00 00
8E70- 00 00 00 00 00 00 00 00
8E78- 00 00 00 00 00 00 00 00
8E80- 00 00 00 00 00 00 00 00
8E88- 00 00 00 00 00 00 00 00
8E90- 00 00 00 00 00 00 00 00
8E98- 00 00 00 00 00 00 00 00
8EA0- 00 00 00 00 00 00 00 00
8EA8- 00 00 00 00 00 00 00 00
8EB0- 00 00 00 00 00 00 00 00
8EB8- 00 00 00 00 00 00 00 00
8EC0- 00 00 00 00 00 00 00 00
8EC8- 00 00 00 00 00 00 00 00
8ED0- 00 00 00 00 00 00 00 00
8ED8- 00 00 00 00 00 00 00 00
8EE0- 00 00 00 00 00 00 00 00
8EE8- 00 00 00 00 00 00 00 00
8EF0- 00 00 00 00 00 00 00 00
8EF8- 00 00 00 00 00 00 00 00
8F00- 00 00 00 00 00 00 00 00
8F08- 00 00 00 00 00 00 00 00
8F10- 00 00 4E 4F 20 53 54 41
8F18- 52 54 20 41 44 44 52 45
8F20- 53 53 00 2D 2D 2D 2D 2D
8F28- 2D 2D 2D 2D 2D 2D 2D 2D
8F30- 2D 2D 2D 2D 2D 2D 2D 20
8F38- 42 52 41 4E 43 48 20 4F
8F40- 55 54 20 4F 46 20 52 41
8F48- 4E 47 45 00 55 4E 44 45
8F50- 46 49 4E 45 44 20 4C 41
8F58- 42 45 4C 00 1D 1D 1D 1D
8F60- 1D 1D 1D 1D 1D 20 4E 41
8F68- 4B 45 44 20 4C 41 42 45
8F70- 4C 00 1D 1D 1D 1D 1D 20
8F78- 3C 3C 3C 3C 3C 3C 3C 3C
8F80- 20 44 49 53 4B 20 45 52
8F88- 52 4F 52 20 3E 3E 3E 3E
8F90- 3E 3E 3E 3E 20 00 1D 1D
8F98- 1D 1D 1D 20 2D 2D 20 44
8FA0- 55 50 4C 49 43 41 54 45
8FA8- 44 20 4C 41 42 45 4C 20
8FB0- 2D 2D 20 00 1D 1D 1D 1D
```

```
8FB8- 1D 20 2D 2D 20 53 59 4E
8FC0- 54 41 58 20 45 52 52 4F
8FC8- 52 20 2D 2D 20 00 00 00
8FD0- 00 00 00 00 00 00 00 00
8FD8- 00 00 00 00 00 00 00 00
8FE0- 00 00 00 00 00 00 00 00
8FE8- 00 00 00 00 00 00 00 00
8FF0- 00 00 00 00 00 00 00 00
8FF8- 00 00 00 00 00 00 00 00
9000- 00 01 00 01 00 00 01 06
9008- 02 2D 93 00 00 00 93 00
9010- 92 00 00 01 00 01 00 00
9018- 01 06 04 80 95 00 00 53
9020- 95 53 94 00 00 03 01 00
9028- 00 00 00 00 00 00 00 00
9030- 00 00 93 00 92 00 91 04
9038- 01 00 00 00 00 00 00 00
9040- 00 00 00 53 95 53 94 53
9048- 93 02 00 00 00 00 00 00
9050- 00 00 00 00 00 00 93 00
9058- 92 00 91 02 00 00 00 00
9060- 00 00 00 00 00 00 00 53
9068- 95 53 94 53 93 00 00 00
9070- 00 01
```

Machine Language Editor for Atari and Commodore

Charles Brannon

Have you ever typed in a long machine language program? Chances are you typed in hundreds of DATA statements, numbers, and commas. You're never sure if you've typed them in right. So you go back, proofread, try to run the program, crash, go back and proofread again, correct a few typing errors, run again, crash, recheck your typing—frustrating, isn't it?

Until now, though, that has been the best way to enter machine language into your computer. Unless you happen to own an assembler and are willing to wrangle with machine language on the assembly level, it is much easier to enter a BASIC program that reads the DATA statements and POKEs the numbers into memory.

Some of these *BASIC loaders*, as they are known, use a *checksum* to see if you've typed the numbers correctly. The simplest checksum is just the sum of all the numbers in the DATA statements. If you make an error, your checksum will not match up. Some programmers make the task easier by calculating checksums every ten lines or so, and you can thereby locate your errors more easily.

Almost Foolproof

"MLX" lets you type in long machine language (ML) listings with almost foolproof results. Using MLX, you enter the numbers from a special list that looks similar to BASIC DATA statements. MLX checks your typing on a line-by-line basis. It won't let you enter illegal characters when you should be typing numbers, such as a lowercase L for a 1 or an O for a 0. It won't let you enter numbers greater than 255, which are not permitted in ML DATA statements. It *will* prevent you from entering the wrong numbers on the wrong line. In short, MLX should make proofreading obsolete!

In addition, MLX will generate a ready-to-use tape or disk file. For the Commodore, you can then use the LOAD command to read the program into the computer, just as you would with any program. Specifically, you enter:

LOAD *"filename"***,1,1** (for tape)

or

LOAD *"filename"***,8,1** (for disk)

To start LADS you need to type SYS 11000 (Commodore). For complete instructions for the use of LADS, please read Appendix A.

For the Atari, MLX will create a binary file for use with DOS. Atari MLX can create a boot disk or tape version of LADS, but this is not recommended.

Getting Started

To get started, type in and save MLX (VIC owners must have at least 8K of extra memory attached). When you are ready to enter LADS using MLX, Commodore 64 and VIC owners should enter the line below before loading MLX:

POKE 55,0: POKE 56,42: CLR

Commodore PET/CBM owners should use:

POKE 52,0: POKE 53,42: CLR

When you're ready to type in LADS, the program will ask you for several numbers: the starting address and the ending address. In addition, the Atari MLX will request a "Run/Init Address".

Below are the numbers you'll need.

PET/CBM, VIC and Commodore 64:
Starting address 11000
Ending address 15985

Atari:
Starting address 32768
Ending address 39607
Run/Init address 32768

The Atari version will then ask you to press either T for a boot tape, or D for disk; *press* D. Next, you'll be asked if you want to generate a boot disk or a binary file; *press* F.

Next you'll see a prompt. The prompt is the current line you are entering from the listing. Each line is six numbers plus a checksum. If you enter any of the six numbers wrong, or enter the checksum wrong, MLX will ring a buzzer and prompt you to reenter the line. If you enter it correctly, a pleasant bell tone will sound and you proceed to the next line.

A Special Editor

You are not using the normal screen editor with MLX. For example, it will accept only numbers as input. If you need to make a correction, press the DEL/BACKS key (Atari) or the INST/DEL key (Commodore). The entire number is deleted. You can press it as many times as necessary back to the start of the line. If you enter three-digit numbers as listed, the computer will automatically print the comma and prepare to accept the next number. If you enter less than three digits (by omitting leading zeros), you can press either the comma, space bar, or RETURN key to advance to the next number. When you get to the checksum value, the Atari MLX will emit a low drone to remind you to be careful. The checksum will automatically appear in inverse video; don't worry, it's highlighted for emphasis.

When testing MLX, we've found that it makes entering long listings extremely easy. We have tested MLX with people lacking any computer background whatsoever. No one here has managed to enter a listing wrong with it.

Done at Last!

When you finish typing (assuming you type the entire listing in one session), you can then save the completed program on tape or disk. Follow the screen instructions. (For Atari we suggest that you use the filename AUTORUN.SYS when saving a copy of LADS. This way LADS will automatically load and run when you boot up your computer.) If you get any errors while saving, you probably have a bad disk, or the disk is full, or you made a typo when entering the actual MLX program. (Remember, it can't check itself!)

Command Control

What if you don't want to enter the whole program in one sitting? MLX lets you enter as much as you want, save that portion, and then reload the file from tape or disk when you want to continue. MLX recognizes these few commands:

S: Save
L: Load

N: New Address
D: Display

For the Atari, hold down the CTRL key while you type the appropriate key. Hold down SHIFT on Commodore machines to enter a command key. You will jump out of the line you've been typing, so it's best to perform these commands at a new prompt. Use the Save command to save what you've been working on. It will write the tape or disk file as if you've finished, but the tape or disk won't work, of course, until you finish the typing. *Remember what address you stop on.* The next time you run MLX, answer all the prompts as you did before, then insert the disk or tape. When you get to the entry prompt, press CTRL-L (Atari) or SHIFT-L (Commodore) to reload the file into memory. You'll then use the New Address command to resume typing.

New Address and Display

Here's how the New Address command works. After you press SHIFT-N or CTRL-N, enter the address where you previously stopped. The prompt will change, and you can then continue typing. Always enter a New Address that matches up with one of the line numbers in the special listing, or else the checksum won't match up.

You can use the Display command to display a section of your typing. After you press CTRL-D or SHIFT-D, enter two addresses within the line number range of the listing. You can abort the listing by pressing any key.

Tricky Business

The special commands may seem a little confusing at first, but as you work with MLX, they will become easy and valuable. What if you forgot where you stopped typing, for instance? Use the Display command to scan memory from the beginning to the end of the program. You can stop a listing by hitting any key.

Making Copies

You can use the MLX Save and Load commands to make copies of the completed ML program. Use Load to reload the tape or disk, then insert a new tape or disk and use the Save command to make a new copy.

PET and VIC Users

The Commodore 64, PET, and VIC data are almost exactly the same. There are some lines, though, that are different. Commodore 64, PET, and VIC owners should use the Commodore 64 data (Program B-1) with MLX. VIC owners should substitute the lines found in Program B-2 (VIC) for the same lines in Program B-1. PET owners should type in and save the 64 data, then make the necessary changes shown in Program B-3a and B-3b using the built-in PET monitor. Commodore 64 users should use the data in Program B-1 as is.

We hope you will find MLX to be a true labor-saving utility. Since it has been thoroughly tested by entering actual programs, you can count on it as an aid for generating bug-free machine language. And be sure to save MLX; it will be used for future all machine language programs in *COMPUTE!*, *COMPUTE!'s Gazette*, and COMPUTE! Books.

Program C-1. Commodore 64 MLX

Refer to Appendix E "How to Type In BASIC Programs" before entering this program.

```
100 PRINT"{CLR}[6]";CHR$(142);CHR$(8);:POKE53281,1
    :POKE53280,1
101 POKE 788,52:REM DISABLE RUN/STOP
110 PRINT"{RVS}{39 SPACES}";
120 PRINT"{RVS}{14 SPACES}{RIGHT}{OFF}[*]£{RVS}
    {RIGHT} {RIGHT}{2 SPACES}[*]{OFF}[*]£{RVS}£
    {RVS}{14 SPACES}";
130 PRINT"{RVS}{14 SPACES}{RIGHT} [G]{RIGHT}
    {2 RIGHT} {OFF}£{RVS}£[*]{OFF}[*]{RVS}
    {14 SPACES}";
140 PRINT"{RVS}{41 SPACES}"
200 PRINT"{2 DOWN}{PUR}{BLK}{9 SPACES}MACHINE LANG
    UAGE EDITOR{5 DOWN}"
210 PRINT"[5]{2 UP}STARTING ADDRESS?{8 SPACES}
    {9 LEFT}";
215 INPUTS:F=1-F:C$=CHR$(31+119*F)
220 IFS<256OR(S>40960ANDS<49152)ORS>53247THENGOSUB
    3000:GOTO210
225 PRINT:PRINT:PRINT
230 PRINT"[5]{2 UP}ENDING ADDRESS?{8 SPACES}
    {9 LEFT}";:INPUTE:F=1-F:C$=CHR$(31+119*F)
240 IFE<256OR(E>40960ANDE<49152)ORE>53247THENGOSUB
    3000:GOTO230
250 IFE<STHENPRINTC$;"{RVS}ENDING < START
    {2 SPACES}":GOSUB1000:GOTO 230
260 PRINT:PRINT:PRINT
300 PRINT"{CLR}";CHR$(14):AD=S:POKEV+21,0
```

```
310 A=1:PRINTRIGHT$("ØØØØ"+MID$(STR$(AD),2),5);":"
    ;
315 FORJ=ATO6
32Ø GOSUB57Ø:IFN=-1THENJ=J+N:GOTO32Ø
39Ø IFN=-211THEN 71Ø
4ØØ IFN=-2Ø4THEN 79Ø
41Ø IFN=-2Ø6THENPRINT:INPUT"{DOWN}ENTER NEW ADDRES
    S";ZZ
415 IFN=-2Ø6THENIFZZ<SORZZ>ETHENPRINT"{RVS}OUT OF
    {SPACE}RANGE":GOSUB1ØØØ:GOTO41Ø
417 IFN=-2Ø6THENAD=ZZ:PRINT:GOTO31Ø
42Ø IF N<>-196 THEN 48Ø
43Ø PRINT?:INPUT"DISPLAY:FROM";F:PRINT,"TO";:INPUTT
44Ø IFF<SORF>EORT<SORT>ETHENPRINT"AT LEAST";S;"
    {LEFT}, NOT MORE THAN";E:GOTO43Ø
45Ø FORI=FTOTSTEP6:PRINT:PRINTRIGHT$("ØØØØ"+MID$(S
    TR$(I),2),5);":";
451 FORK=ØTO5:N=PEEK(I+K):PRINTRIGHT$("ØØ"+MID$(ST
    R$(N),2),3);",";
46Ø GETA$:IFA$>""THENPRINT:PRINT:GOTO31Ø
47Ø NEXTK:PRINTCHR$(2Ø);:NEXTI:PRINT:PRINT:GOTO31Ø
48Ø IFN<Ø THEN PRINT:GOTO31Ø
49Ø A(J)=N:NEXTJ
5ØØ CKSUM=AD-INT(AD/256)*256:FORI=1TO6:CKSUM=(CKSU
    M+A(I))AND255:NEXT
51Ø PRINTCHR$(18);:GOSUB57Ø:PRINTCHR$(146);
511 IFN=-1THENA=6:GOTO315
515 PRINTCHR$(2Ø):IFN=CKSUMTHEN53Ø
52Ø PRINT:PRINT"LINE ENTERED WRONG : RE-ENTER":PRI
    NT:GOSUB1ØØØ:GOTO31Ø
53Ø GOSUB2ØØØ
54Ø FORI=1TO6:POKEAD+I-1,A(I):NEXT:POKE54272,Ø:POK
    E54273,Ø
55Ø AD=AD+6:IF AD<E THEN 31Ø
56Ø GOTO 71Ø
57Ø N=Ø:Z=Ø
58Ø PRINT"[£]";
581 GETA$:IFA$=""THEN581
585 PRINTCHR$(2Ø);:A=ASC(A$):IFA=13ORA=44ORA=32THE
    N67Ø
59Ø IFA>128THENN=-A:RETURN
6ØØ IFA<>2Ø THEN 63Ø
61Ø GOSUB69Ø:IFI=1ANDT=44THENN=-1:PRINT"{OFF}
    {LEFT} {LEFT}";:GOTO69Ø
62Ø GOTO57Ø
63Ø IFA<48ORA>57THEN58Ø
64Ø PRINTA$;:N=N*1Ø+A-48
65Ø IFN>255 THEN A=2Ø:GOSUB1ØØØ:GOTO6ØØ
66Ø Z=Z+1:IFZ<3THEN58Ø
```

```
670 IFZ=0THENGOSUB1000:GOTO570
680 PRINT",";:RETURN
690 S%=PEEK(209)+256*PEEK(210)+PEEK(211)
691 FORI=1TO3:T=PEEK(S%-I)
695 IFT<>44ANDT<>58THENPOKES%-I,32:NEXT
700 PRINTLEFT$("{3 LEFT}",I-1);:RETURN
710 PRINT"{CLR}{RVS}*** SAVE ***{3 DOWN}"
715 PRINT"{2 DOWN}(PRESS {RVS}RETURN{OFF} ALONE TO
     CANCEL SAVE){DOWN}"
720 F$="":INPUT"{DOWN} FILENAME";F$:IFF$=""THENPRI
    NT:PRINT:GOTO310
730 PRINT:PRINT"{2 DOWN}{RVS}T{OFF}APE OR {RVS}D
    {OFF}ISK: (T/D)"
740 GETA$:IFA$<>"T"ANDA$<>"D"THEN740
750 DV=1-7*(A$="D"):IFDV=8THENF$="0:"+F$:OPEN15,8,
    15,"S"+F$:CLOSE15
760 T$=F$:ZK=PEEK(53)+256*PEEK(54)-LEN(T$):POKE782
    ,ZK/256
762 POKE781,ZK-PEEK(782)*256:POKE780,LEN(T$):SYS65
    469
763 POKE780,1:POKE781,DV:POKE782,1:SYS65466
765 K=S:POKE254,K/256:POKE253,K-PEEK(254)*256:POKE
    780,253
766 K=E+1:POKE782,K/256:POKE781,K-PEEK(782)*256:SY
    S65496
770 IF(PEEK(783)AND1)OR(191ANDST)THEN780
775 PRINT"{DOWN}DONE.{DOWN}":GOTO310
780 PRINT"{DOWN}ERROR ON SAVE.{2 SPACES}TRY AGAIN.
    ":IFDV=1THEN720
781 OPEN15,8,15:INPUT#15,E1$,E2$:PRINTE1$;E2$:CLOS
    E15:GOTO720
790 PRINT"{CLR}{RVS}*** LOAD ***{2 DOWN}"
795 PRINT"{2 DOWN}(PRESS {RVS}RETURN{OFF} ALONE TO
     CANCEL LOAD)"
800 F$="":INPUT"{2 DOWN} FILENAME";F$:IFF$=""THENP
    RINT:GOTO310
810 PRINT:PRINT"{2 DOWN}{RVS}T{OFF}APE OR {RVS}D
    {OFF}ISK: (T/D)"
820 GETA$:IFA$<>"T"ANDA$<>"D"THEN820
830 DV=1-7*(A$="D"):IFDV=8THENF$="0:"+F$
840 T$=F$:ZK=PEEK(53)+256*PEEK(54)-LEN(T$):POKE782
    ,ZK/256
841 POKE781,ZK-PEEK(782)*256:POKE780,LEN(T$):SYS65
    469
845 POKE780,1:POKE781,DV:POKE782,1:SYS65466
850 POKE780,0:SYS65493
860 IF(PEEK(783)AND1)OR(191ANDST)THEN870
865 PRINT"{DOWN}DONE.":GOTO310
```

```
870 PRINT"{DOWN}ERROR ON LOAD.{2 SPACES}TRY AGAIN.
    {DOWN}":IFDV=1THEN8ØØ
88Ø OPEN15,8,15:INPUT#15,E1$,E2$:PRINTE1$;E2$:CLOS
    E15:GOTO8ØØ
1ØØØ REM BUZZER
1ØØ1 POKE54296,15:POKE54277,45:POKE54278,165
1ØØ2 POKE54276,33:POKE 54273,6:POKE54272,5
1ØØ3 FORT=1TO2ØØ:NEXT:POKE54276,32:POKE54273,Ø:POK
     E54272,Ø:RETURN
2ØØØ REM BELL SOUND
2ØØ1 POKE54296,15:POKE54277,Ø:POKE54278,247
2ØØ2 POKE 54276,17:POKE54273,4Ø:POKE54272,Ø
2ØØ3 FORT=1TO1ØØ:NEXT:POKE54276,16:RETURN
3ØØØ PRINTC$;"{RVS}NOT ZERO PAGE OR ROM":GOTO1ØØØ
```

Program C-2. VIC MLX

Refer to Appendix E "How to Type In BASIC Programs" before entering this program.

```
1ØØ PRINT"{CLR}{PUR}";CHR$(142);CHR$(8);
1Ø1 POKE 788,194:REM DISABLE RUN/STOP
11Ø PRINT"{RVS}{14 SPACES}"
12Ø PRINT"{RVS} {RIGHT}{OFF}[*]£{RVS}{RIGHT}
    {RIGHT}{2 SPACES}[*]{OFF}[*]£{RVS}£{RVS} "
13Ø PRINT"{RVS} {RIGHT} [G]{RIGHT} {2 RIGHT} {OFF}
    £{RVS}£[*]{OFF}[*]{RVS} "
14Ø PRINT"{RVS}{14 SPACES}"
2ØØ PRINT"{2 DOWN}{PUR}{BLK}A FAILSAFE MACHINE":PR
    INT"LANGUAGE EDITOR{5 DOWN}"
21Ø PRINT"{BLK}{3 UP}STARTING ADDRESS":INPUTS:F=1-
    F:C$=CHR$(31+119*F)
22Ø IFS<256ORS>32767THENGOSUB3ØØØ:GOTO21Ø
225 PRINT:PRINT:PRINT:PRINT
23Ø PRINT"{BLK}{3 UP}ENDING ADDRESS":INPUTE:F=1-F:
    C$=CHR$(31+119*F)
24Ø IFE<256ORE>32767THENGOSUB3ØØØ:GOTO23Ø
25Ø IFE<STHENPRINTC$;"{RVS}ENDING < START
    {2 SPACES}":GOSUB1ØØØ:GOTO 23Ø
26Ø PRINT:PRINT:PRINT
3ØØ PRINT"{CLR}";CHR$(14):AD=S
31Ø PRINTRIGHT$("ØØØØ"+MID$(STR$(AD),2),5);":";:FO
    RJ=1TO6
32Ø GOSUB57Ø:IFN=-1THENJ=J+N:GOTO32Ø
39Ø IFN=-211THEN 71Ø
4ØØ IFN=-2Ø4THEN 79Ø
41Ø IFN=-2Ø6THENPRINT:INPUT"{DOWN}ENTER NEW ADDRES
    S";ZZ
415 IFN=-2Ø6THENIFZZ<SORZZ>ETHENPRINT"{RVS}OUT OF
    {SPACE}RANGE":GOSUB1ØØØ:GOTO41Ø
```

```
417 IFN=-2Ø6THENAD=ZZ:PRINT:GOTO31Ø
42Ø IF N<>-196 THEN 48Ø
43Ø PRINT:INPUT"DISPLAY:FROM";F:PRINT,"TO";:INPUTT
44Ø IFF<SORF>EORT<SORT>ETHENPRINT"AT LEAST";S;"
    {LEFT}, NOT MORE THAN";E:GOTO43Ø
45Ø FORI=FTOTSTEP6:PRINT:PRINTRIGHT$("ØØØØ"+MID$(S
    TR$(I),2),5);":";
455 FORK=ØTO5:N=PEEK(I+K):IFK=3THENPRINTSPC(1Ø);
457 PRINTRIGHT$("ØØ"+MID$(STR$(N),2),3);",";
46Ø GETA$:IFA$>""THENPRINT:PRINT:GOTO31Ø
47Ø NEXTK:PRINTCHR$(2Ø);:NEXTI:PRINT:PRINT:GOTO31Ø
48Ø IFN<Ø THEN PRINT:GOTO31Ø
49Ø A(J)=N:NEXTJ
5ØØ CKSUM=AD-INT(AD/256)*256:FORI=1TO6:CKSUM=(CKSU
    M+A(I))AND255:NEXT
51Ø PRINTCHR$(18);:GOSUB57Ø:PRINTCHR$(2Ø)
515 IFN=CKSUMTHEN53Ø
52Ø PRINT:PRINT"LINE ENTERED WRONG":PRINT"RE-ENTER
    ":PRINT:GOSUB1ØØØ:GOTO31Ø
53Ø GOSUB2ØØØ
54Ø FORI=1TO6:POKEAD+I-1,A(I):NEXT
55Ø AD=AD+6:IF AD<E THEN 31Ø
56Ø GOTO 71Ø
57Ø N=Ø:Z=Ø
58Ø PRINT"[+]";
581 GETA$:IFA$=""THEN581
585 PRINTCHR$(2Ø);:A=ASC(A$):IFA=13ORA=44ORA=32THE
    N67Ø
59Ø IFA>128THENN=-A:RETURN
6ØØ IFA<>2Ø THEN 63Ø
61Ø GOSUB69Ø:IFI=1ANDT=44THENN=-1:PRINT"{LEFT}
    {LEFT}";:GOTO69Ø
62Ø GOTO57Ø
63Ø IFA<48ORA>57THEN58Ø
64Ø PRINTA$;:N=N*1Ø+A-48
65Ø IFN>255 THEN A=2Ø:GOSUB1ØØØ:GOTO6ØØ
66Ø Z=Z+1:IFZ<3THEN58Ø
67Ø IFZ=ØTHENGOSUB1ØØØ:GOTO57Ø
68Ø PRINT",";:RETURN
69Ø S%=PEEK(2Ø9)+256*PEEK(21Ø)+PEEK(211)
692 FORI=1TO3:T=PEEK(S%-I)
695 IFT<>44ANDT<>58THENPOKES%-I,32:NEXT
7ØØ PRINTLEFT$("{3 LEFT}",I-1);:RETURN
71Ø PRINT"{CLR}{RVS}*** SAVE ***{3 DOWN}"
72Ø INPUT"{DOWN} FILENAME";F$
73Ø PRINT:PRINT"{2 DOWN}{RVS}T{OFF}APE OR {RVS}D
    {OFF}ISK: (T/D)"
```

```
740 GETA$:IFA$<>"T"ANDA$<>"D"THEN740
750 DV=1-7*(A$="D"):IFDV=8THENF$="0:"+F$
760 T$=F$:ZK=PEEK(53)+256*PEEK(54)-LEN(T$):POKE782
    ,ZK/256
762 POKE781,ZK-PEEK(782)*256:POKE780,LEN(T$):SYS65
    469
763 POKE780,1:POKE781,DV:POKE782,1:SYS65466
765 POKE254,S/256:POKE253,S-PEEK(254)*256:POKE780,
    253
766 POKE782,E/256:POKE781,E-PEEK(782)*256:SYS65496
770 IF(PEEK(783)AND1)OR(ST AND191)THEN780
775 PRINT"{DOWN}DONE.":END
780 PRINT"{DOWN}ERROR ON SAVE.{2 SPACES}TRY AGAIN.
    ":IFDV=1THEN720
781 OPEN15,8,15:INPUT#15,E1$,E2$:PRINTE1$;E2$:CLOS
    E15:GOTO720
782 GOTO720
790 PRINT"{CLR}{RVS}*** LOAD ***{2 DOWN}"
800 INPUT"{2 DOWN} FILENAME";F$
810 PRINT:PRINT"{2 DOWN}{RVS}T{OFF}APE OR {RVS}D
    {OFF}ISK: (T/D)"
820 GETA$:IFA$<>"T"ANDA$<>"D"THEN820
830 DV=1-7*(A$="D"):IFDV=8THENF$="0:"+F$
840 T$=F$:ZK=PEEK(53)+256*PEEK(54)-LEN(T$):POKE782
    ,ZK/256
841 POKE781,ZK-PEEK(782)*256:POKE780,LEN(T$):SYS65
    469
845 POKE780,1:POKE781,DV:POKE782,1:SYS65466
850 POKE780,0:SYS65493
860 IF(PEEK(783)AND1)OR(ST AND191)THEN870
865 PRINT"{DOWN}DONE.":GOTO310
870 PRINT"{DOWN}ERROR ON LOAD.{2 SPACES}TRY AGAIN.
    {DOWN}":IFDV=1THEN800
880 OPEN15,8,15:INPUT#15,E1$,E2$:PRINTE1$;E2$:CLOS
    E15:GOTO800
1000 REM BUZZER
1001 POKE36878,15:POKE36874,190
1002 FORW=1TO300:NEXTW
1003 POKE36878,0:POKE36874,0:RETURN
2000 REM BELL SOUND
2001 FORW=15TO0STEP-1:POKE36878,W:POKE36876,240:NE
     XTW
2002 POKE36876,0:RETURN
3000 PRINTC$;"{RVS}NOT ZERO PAGE OR ROM":GOTO1000
```

Program C-3. PET MLX

Refer to Appendix E "How to Type In BASIC Programs" before entering this program.

```
100 PRINT"{CLR}";CHR$(142):POKE53,43:CLR
110 PRINT"{RVS}{38 SPACES}"
120 PRINT"{RVS}{18 SPACES}MLX{17 SPACES}"
140 PRINT"{RVS}{38 SPACES}"
200 PRINT"{2 DOWN} MACHINE LANGUAGE EDITOR PET VER
    SION{5 DOWN}"
210 PRINT"{2 UP}STARTING ADDRESS?{8 SPACES}
    {9 LEFT}";
215 INPUTS
220 IFS<256ORS>32767THENGOSUB3000:GOTO210
225 PRINT:PRINT:PRINT
230 PRINT"{2 UP}ENDING ADDRESS?{8 SPACES}{9 LEFT}"
    ;:INPUTE
240 IFE<256ORE>32767THENGOSUB3000:GOTO230
250 IFE<STHENPRINTC$;"{RVS}ENDING < START
    {2 SPACES}":GOSUB1000:GOTO 230
260 PRINT:PRINT:PRINT
300 PRINT"{CLR}";CHR$(14):AD=S
310 A=1:PRINTRIGHT$("0000"+MID$(STR$(AD),2),5);":"
    ;
315 FORJ=ATO6
320 GOSUB570:IFN=-1THENJ=J+N:GOTO320
390 IFN=-211THEN 710
400 IFN=-204THEN 790
410 IFN=-206THENPRINT:INPUT"{DOWN}ENTER NEW ADDRES
    S";ZZ
415 IFN=-206THENIFZZ<SORZZ>ETHENPRINT"{RVS}OUT OF
    {SPACE}RANGE":GOSUB1000:GOTO410
417 IFN=-206THENAD=ZZ:PRINT:GOTO310
420 IF N<>-196 THEN 480
430 PRINT:INPUT"DISPLAY:FROM";F:PRINT,"TO";:INPUTT
440 IFF<SORF>EORT<SORT>ETHENPRINT"AT LEAST";S;"
    {LEFT}, NOT MORE THAN";E:GOTO430
450 FORI=FTOTSTEP6:PRINT:PRINTRIGHT$("0000"+MID$(S
    TR$(I),2),5);":";
451 FORK=0TO5:N=PEEK(I+K):PRINTRIGHT$("00"+MID$(ST
    R$(N),2),3);",";
460 GETA$:IFA$>""THENPRINT:PRINT:GOTO310
470 NEXTK:PRINTCHR$(20);:NEXTI:PRINT:PRINT:GOTO310
480 IFN<0 THEN PRINT:GOTO310
490 A(J)=N:NEXTJ
500 CKSUM=AD-INT(AD/256)*256:FORI=1TO6:CKSUM=(CKSU
    M+A(I))AND255:NEXT
510 PRINTCHR$(18);:GOSUB570:PRINTCHR$(146);
511 IFN=-1THENA=6:GOTO315
515 PRINTCHR$(20):IFN=CKSUMTHEN530
```

```
520 PRINT:PRINT"LINE ENTERED WRONG : RE-ENTER":PRI
    NT:GOSUB1000:GOTO310
530 GOSUB2000
540 FORI=1TO6:POKEAD+I-1,A(I):NEXT
550 AD=AD+6:IF AD<E THEN 310
560 GOTO 710
570 N=0:Z=0
580 PRINTCHR$(168);
581 GETA$:IFA$=""THEN581
585 PRINTCHR$(20);:A=ASC(A$):IFA=13ORA=44ORA=32THE
    N670
590 IFA>128THENN=-A:RETURN
600 IFA<>20 THEN 630
610 GOSUB690:IFI=1ANDT=44THENN=-1:PRINT"{OFF}
    {LEFT} {LEFT}";:GOTO690
620 GOTO570
630 IFA<48ORA>57THEN580
640 PRINTA$;:N=N*10+A-48
650 IFN>255 THEN A=20:GOSUB1000:GOTO600
660 Z=Z+1:IFZ<3THEN580
670 IFZ=0THENGOSUB1000:GOTO570
680 PRINT",";:RETURN
690 SS=PEEK(196)+256*PEEK(197)+PEEK(198)
691 FORI=1TO3:T=PEEK(SS-I)
695 IFT<>44ANDT<>58THENPOKESS-I,32:NEXT
700 PRINTLEFT$("{3 LEFT}",I-1);:RETURN
710 PRINT"{CLR}{RVS}*** SAVE ***{3 DOWN}"
715 PRINT"{2 DOWN}(PRESS {RVS}RETURN{OFF} ALONE TO
     CANCEL SAVE){DOWN}"
720 F$="":INPUT"{DOWN} FILENAME? *{3 LEFT}";F$:IFF
    $="*"THENPRINT:PRINT:GOTO310
730 PRINT:PRINT"{2 DOWN}{RVS}T{OFF}APE OR {RVS}D
    {OFF}ISK: (T/D)"
740 GETA$:IFA$<>"T"ANDA$<>"D"THEN740
750 DV=1-7*(A$="D"):IFDV=8THENF$="0:"+F$:OPEN15,8,
    15,"S"+F$:CLOSE15
760 T$=F$:ZK=PEEK(50)+256*PEEK(51)-LEN(T$):POKE219
    ,ZK/256
762 POKE218,ZK-PEEK(219)*256:POKE209,LEN(T$)
763 POKE210,1:POKE211,0:POKE212,DV
765 K=S:POKE252,K/256:POKE251,K-PEEK(252)*256
766 K=E+1:POKE202,K/256:POKE201,K-PEEK(202)*256:SY
    S63203:REM 63140 FOR 3.0
770 IF(191ANDST)THEN780
775 PRINT"{DOWN}DONE.{DOWN}":GOTO310
780 PRINT"{DOWN}ERROR ON SAVE.{2 SPACES}TRY AGAIN.
    ":IFDV=1THEN720
781 OPEN15,8,15:INPUT#15,E1$,E2$:PRINTE1$;E2$:CLOS
    E15:GOTO720
```

```
790 PRINT"{CLR}{RVS}*** LOAD ***{2 DOWN}"
795 PRINT"{2 DOWN}(PRESS {RVS}RETURN{OFF} ALONE TO
    CANCEL LOAD)"
8ØØ F$="":INPUT"{2 DOWN} FILENAME? *{3 LEFT}";F$:I
   FF$="*"THENPRINT:PRINT:GOTO31Ø
81Ø PRINT:PRINT"{2 DOWN}{RVS}T{OFF}APE OR {RVS}D
   {OFF}ISK: (T/D)"
82Ø GETA$:IFA$<>"T"ANDA$<>"D"THEN82Ø
83Ø DV=1-7*(A$="D"):IFDV=8THENF$="Ø:"+F$
84Ø T$=F$:ZK=PEEK(5Ø)+256*PEEK(51)-LEN(T$):POKE219
   ,ZK/256
841 POKE218,ZK-PEEK(219)*256:POKE2Ø9,LEN(T$)
845 POKE21Ø,1:POKE211,Ø:POKE212,DV
85Ø POKE157,Ø:SYS62294:REM USE 62242 FOR UPGRADE P
   ET 3.Ø
86Ø IF(191ANDST)THEN87Ø
865 PRINT"{DOWN}DONE.":GOTO31Ø
87Ø PRINT"{DOWN}ERROR ON LOAD.{2 SPACES}TRY AGAIN.
   {DOWN}":IFDV=1THEN8ØØ
88Ø OPEN15,8,15:INPUT#15,E1$,E2$:PRINTE1$;E2$:CLOS
   E15:GOTO8ØØ
1ØØØ REM BUZZER
1ØØ1 POKE59467,16:POKE59466,129:POKE59464,255
1ØØ3 FORT=2ØØTO25Ø:POKE59466,T:NEXT:POKE59467,Ø:RE
     TURN
2ØØØ REM BELL SOUND
2ØØ1 POKE59467,16:POKE59466,51:POKE59464,1ØØ
2ØØ3 FORT=1TO5Ø:NEXT:POKE59467,Ø:RETURN
3ØØØ PRINT"{RVS}NOT ZERO PAGE, SCREEN OR ROM":GOTO
     1ØØØ
```

Program C-4. Atari MLX

Refer to Appendix E "How to Type In BASIC Programs" before entering this program.

```
1ØØ GRAPHICS Ø:DL=PEEK(56Ø)+256*PEEK(561)+4:
    POKE DL-1,71:POKE DL+2,6
11Ø POSITION 8,Ø:? "MLX":POSITION 23,Ø:? "fa
    ilsafe entry":POKE 71Ø,Ø:?
12Ø ? "Starting Address";:INPUT BEG:? "  End
    ing Address";:INPUT FIN:? "Run/Init Addr
    ess";:INPUT STARTADR
13Ø DIM A(6),BUFFER$(FIN-BEG+127),T$(2Ø),F$(
    2Ø),CIO$(7),SECTOR$(128),DSKINV$(6)
14Ø OPEN #1,4,Ø,"K:":? :? ,"Tape or Disk:";
15Ø BUFFER$=CHR$(Ø):BUFFER$(FIN-BEG+3Ø)=BUFF
    ER$:BUFFER$(2)=BUFFER$:SECTOR$=BUFFER$
16Ø ADDR=BEG:CIO$="hhh":CIO$(4)=CHR$(17Ø):CI
    O$(5)="LV":CIO$(7)=CHR$(228)
17Ø GET #1,MEDIA:IF MEDIA<>84 AND MEDIA<>68
    THEN 17Ø
```

```
180 ? CHR$(MEDIA):? :IF MEDIA<>ASC("T") THEN BUFFER$="":GOTO 250
190 BEG=BEG-24:BUFFER$=CHR$(0):BUFFER$(2)=CHR$((FIN-BEG+127)/128)
200 H=INT(BEG/256):L=BEG-H*256:BUFFER$(3)=CHR$(L):BUFFER$(4)=CHR$(H)
210 PINIT=BEG+8:H=INT(PINIT/256):L=PINIT-H*256:BUFFER$(5)=CHR$(L):BUFFER$(6)=CHR$(H)
220 FOR I=7 TO 24:READ A:BUFFER$(I)=CHR$(A):NEXT I:DATA 24,96,169,60,141,2,211,169,0,133,10,169,0,133,11,76,0,0
230 H=INT(STARTADR/256):L=STARTADR-H*256:BUFFER$(15)=CHR$(L):BUFFER$(19)=CHR$(H)
240 BUFFER$(23)=CHR$(L):BUFFER$(24)=CHR$(H)
250 IF MEDIA<>ASC("D") THEN 360
260 ? :? "Boot Disk or Binary File:";
270 GET #1,DTYPE:IF DTYPE<>68 AND DTYPE<>70 THEN 270
280 ? CHR$(DTYPE):IF DTYPE=70 THEN 360
290 BEG=BEG-30:BUFFER$=CHR$(0):BUFFER$(2)=CHR$((FIN-BEG+127)/128)
300 H=INT(BEG/256):L=BEG-H*256:BUFFER$(3)=CHR$(L):BUFFER$(4)=CHR$(H)
310 PINIT=STARTADR:H=INT(PINIT/256):L=PINIT-H*256:BUFFER$(5)=CHR$(L):BUFFER$(6)=CHR$(H)
320 RESTORE 330:FOR I=7 TO 30:READ A:BUFFER$(I)=CHR$(A):NEXT I
330 DATA 169,0,141,231,2,133,14,169,0,141,232,2,133,15,169,0,133,10,169,0,133,11,24,96
340 H=INT(BEG/256):L=BEG-H*256:BUFFER$(8)=CHR$(L):BUFFER$(15)=CHR$(H)
350 H=INT(STARTADR/256):L=STARTADR-H*256:BUFFER$(22)=CHR$(L):BUFFER$(26)=CHR$(H)
360 GRAPHICS 0:POKE 712,10:POKE 710,10:POKE 709,2
370 ? ADDR;":";:FOR J=1 TO 6
380 GOSUB 570:IF N=-1 THEN J=J-1:GOTO 380
390 IF N=-19 THEN 720
400 IF N=-12 THEN LET READ=1:GOTO 720
410 TRAP 410:IF N=-14 THEN ? :? "New Address ";:INPUT ADDR:? :GOTO 370
420 TRAP 32767:IF N<>-4 THEN 480
430 TRAP 430:? :? "Display:From";:INPUT F:?,"To";:INPUT T:TRAP 32767
440 IF F<BEG OR F>FIN OR T<BEG OR T>FIN OR T<F THEN ? CHR$(253);"At least ";BEG;", Not More Than ";FIN:GOTO 430
```

```
450 FOR I=F TO T STEP 6:? :? I;":";:FOR K=0 TO 5:N=PEEK(ADR(BUFFER$)+I+K-BEG):T$="000":T$(4-LEN(STR$(N)))=STR$(N)
460 IF PEEK(764)<255 THEN GET #1,A:POP :POP :? :GOTO 370
470 ? T$;",";:NEXT K:? CHR$(126);:NEXT I:? :? :GOTO 370
480 IF N<0 THEN ? :GOTO 370
490 A(J)=N:NEXT J
500 CKSUM=ADDR-INT(ADDR/256)*256:FOR I=1 TO 6:CKSUM=CKSUM+A(I):CKSUM=CKSUM-256*(CKSUM>255):NEXT I
510 RF=128:SOUND 0,200,12,8:GOSUB 570:SOUND 0,0,0,0:RF=0:? CHR$(126)
520 IF N<>CKSUM THEN ? :? "Incorrect";CHR$(253);:? :GOTO 370
530 FOR W=15 TO 0 STEP -1:SOUND 0,50,10,W:NEXT W
540 FOR I=1 TO 6:POKE ADR(BUFFER$)+ADDR-BEG+I-1,A(I):NEXT I
550 ADDR=ADDR+6:IF ADDR<=FIN THEN 370
560 GOTO 710
570 N=0:Z=0
580 GET #1,A:IF A=155 OR A=44 OR A=32 THEN 670
590 IF A<32 THEN N=-A:RETURN
600 IF A<>126 THEN 630
610 GOSUB 690:IF I=1 AND T=44 THEN N=-1:? CHR$(126);:GOTO 690
620 GOTO 570
630 IF A<48 OR A>57 THEN 580
640 ? CHR$(A+RF);:N=N*10+A-48
650 IF N>255 THEN ? CHR$(253);:A=126:GOTO 600
660 Z=Z+1:IF Z<3 THEN 580
670 IF Z=0 THEN ? CHR$(253);:GOTO 570
680 ? ",";:RETURN
690 POKE 752,1:FOR I=1 TO 3:? CHR$(30);:GET #6,T:IF T<>44 AND T<>58 THEN ? CHR$(A);:NEXT I
700 POKE 752,0:? " ";CHR$(126);:RETURN
710 GRAPHICS 0:POKE 710,26:POKE 712,26:POKE 709,2
720 IF MEDIA=ASC("T") THEN 890
730 REM DISK
740 IF READ THEN ? :? "Load File":?
750 IF DTYPE<>ASC("F") THEN 1040
760 ? :? "Enter AUTORUN.SYS for automatic use":? :? "Enter filename":INPUT T$
```

```
770 F$=T$:IF LEN(T$)>2 THEN IF T$(1,2)<>"D:"
     THEN F$="D:":F$(3)=T$
780 TRAP 870:CLOSE #2:OPEN #2,8-4*READ,0,F$:
    ? :? "Working..."
790 IF READ THEN FOR I=1 TO 6:GET #2,A:NEXT
    I:GOTO 820
800 PUT #2,255:PUT #2,255
810 H=INT(BEG/256):L=BEG-H*256:PUT #2,L:PUT
    #2,H:H=INT(FIN/256):L=FIN-H*256:PUT #2,L
    :PUT #2,H
820 GOSUB 970:IF PEEK(195)>1 THEN 870
830 IF STARTADR=0 OR READ THEN 850
840 PUT #2,224:PUT #2,2:PUT #2,225:PUT #2,2:
    H=INT(STARTADR/256):L=STARTADR-H*256:PUT
     #2,L:PUT #2,H
850 TRAP 32767:CLOSE #2:? "Finished.":IF REA
    D THEN ? :? :LET READ=0:GOTO 360
860 END
870 ? "Error ";PEEK(195);" trying to access"
    :? F$:CLOSE #2:? :GOTO 760
880 REM  BOOT TAPE
890 IF READ THEN ? :? "Read Tape"
900 ? :? :? "Insert, Rewind Tape.":? "Press
    PLAY ";:IF  NOT READ THEN ? "& RECORD"
910 ? :? "Press RETURN when ready:";
920 TRAP 960:CLOSE #2:OPEN #2,8-4*READ,128,"
    C:":? :? "Working..."
930 GOSUB 970:IF PEEK(195)>1 THEN 960
940 CLOSE #2:TRAP 32767:? "Finished.":? :? :
    IF READ THEN LET READ=0:GOTO 360
950 END
960 ? :? "Error ";PEEK(195);" when reading/w
    riting boot tape":? :CLOSE #2:GOTO 890
970 REM CIO Load/Save File#2 opened  READ=0
    for write, READ=1 for read
980 X=32:REM File#2,$20
990 ICCOM=834:ICBADR=836:ICBLEN=840:ICSTAT=8
    35
1000 H=INT(ADR(BUFFER$)/256):L=ADR(BUFFER$)-
     H*256:POKE ICBADR+X,L:POKE ICBADR+X+1,H
1010 L=FIN-BEG+1:H=INT(L/256):L=L-H*256:POKE
      ICBLEN+X,L:POKE ICBLEN+X+1,H
1020 POKE ICCOM+X,11-4*READ:A=USR(ADR(CIO$),
     X)
1030 POKE 195,PEEK(ICSTAT):RETURN
1040 REM  SECTOR I/O
1050 IF READ THEN 1100
1060 ? :? "Format Disk In Drive 1? (Y/N):";
```

```
1070 GET #1,A:IF A<>78 AND A<>89 THEN 1070
1080 ? CHR$(A):IF A=78 THEN 1100
1090 ? :? "Formatting...":XIO 254,#2,0,0,"D:
     ":? "Format Complete":?
1100 NR=INT((FIN-BEG+127)/128):BUFFER$(FIN-B
     EG+2)=CHR$(0):IF READ THEN ? "Reading..
     .":GOTO 1120
1110 ? "Writing..."
1120 FOR I=1 TO NR:S=I
1130 IF READ THEN GOSUB 1220:BUFFER$(I*128-1
     27)=SECTOR$:GOTO 1160
1140 SECTOR$=BUFFER$(I*128-127)
1150 GOSUB 1220
1160 IF PEEK(DSTATS)<>1 THEN 1200
1170 NEXT I
1180 IF  NOT READ THEN END
1190 ? :? :LET READ=0:GOTO 360
1200 ? "Error on disk access.":? "May need f
     ormatting.":GOTO 1040
1210 REM
1220 REM  SECTOR ACCESS SUBROUTINE
1230 REM Drive ONE
1240 REM Pass buffer in SECTOR$
1250 REM sector # in variable S
1260 REM READ=1 for read,
1270 REM READ=0 for write
1280 BASE=3*256
1290 DUNIT=BASE+1:DCOMND=BASE+2:DSTATS=BASE+
     3
1300 DBUFLO=BASE+4:DBUFHI=BASE+5
1310 DBYTLO=BASE+8:DBYTHI=BASE+9
1320 DAUX1=BASE+10:DAUX2=BASE+11
1330 REM DIM DSKINV$(4)
1340 DSKINV$="hLS":DSKINV$(4)=CHR$(228)
1350 POKE DUNIT,1:A=ADR(SECTOR$):H=INT(A/256
     ):L=A-256*H
1360 POKE DBUFHI,H
1370 POKE DBUFLO,L
1380 POKE DCOMND,87-5*READ
1390 POKE DAUX2,INT(S/256):POKE DAUX1,S-PEEK
     (DAUX2)*256
1400 A=USR(ADR(DSKINV$))
1410 RETURN
```

A Library of Subroutines

Here is a collection of techniques you'll need to use in many of your ML programs. Those techniques which are not inherently easy to understand are followed by an explanation.

Increment and Decrement Double-Byte Numbers

You'll often want to raise or lower a number by 1. To *increment* a number, you add 1 to it: Incrementing 5 results in 6. Decrement lowers a number by 1. Single-byte numbers are easy; you just use INC or DEC. But you'll often want to increment two-byte numbers which hold addresses, game scores, pointers, or some other number which requires two bytes. Two bytes, ganged together and seen as a single number, can hold values from 0 ($0000) up to 65535 ($FFFF). Here's how to raise a two-byte number by 1, to increment it:

(Let's assume that the number you want to increment or decrement is located in addresses $0605 and $0606, and the ML program segment performing the action is located at $5000.)

```
5000 INCREMENT INC $0605; raise the low byte
5003 BNE GOFORTH; if not zero, leave high byte alone
5005 INC $0606; raise high byte
5008 GOFORTH ... continue with program
```

The trick in this routine is the BNE. If the low byte isn't raised to zero (from 255), we don't need to add a "carry" to the high byte, so we jump over it. However, if the low byte does turn into a zero, the high byte must then be raised. This is similar to the way an ordinary decimal increment creates a carry when you add 1 to 9 (or 99 or 999). The lower number turns to zero, and the next column over is raised by one.

To double decrement, you need an extra step. The reason it's more complicated is that the 6502 chip has no way to test if you've crossed over to $FF, down from $00. BNE and BEQ will test if something is zero, but nothing tests for $FF. (The N flag *is* turned on when you go from $00 to $FF, and BPL or BMI could test it.) The problem with it, though, is that the N flag isn't limited to sensing $FF. It is sensitive to *any* number higher than 127 decimal ($7F).

So, here's the way to handle double-deckers:

5000 LDA $0605; load in the low byte (affecting the zero flag)
5003 BNE FIXLOWBYTE; if it's not zero, lower it, skipping high byte
5005 DEC $0606; zero in low byte forces this.
5008 FIXLOWBYTE DEC $0605; always dec the low byte.

Here we *always* lower the low byte, but lower the high byte only when the low byte is found to be zero. If you think about it, that's the way any subtraction would work.

Comparison

Comparing a single-byte against another single-byte is easily achieved with CMP. Double-byte comparison can be handled this way:

(Assume that the numbers you want to compare are located in addresses $0605,0606 and $0700,0701. The ML program segment performing the comparison is located at $5000.)

5000 SEC
5001 LDA $0605; low byte of first number
5004 SBC $0700; low byte of second number
5007 STA $0800; temporary holding place for this result
500A LDA $0606; high byte of first number
500D SBC $0701; high byte of second number, leave result in A
5010 ORA $0800; results in zero if A and $0800 were both zero.

The flags in the Status Register are left in various states after this routine—you can test them with the B instructions and branch according to the results. The ORA sets the Z (zero) flag if the results of the first subtraction (left in $0800) and the second subtraction (in A, the Accumulator) were both zero. This would only happen if the two numbers tested were identical, and BEQ would test for this (Branch if EQual).

If the first number is lower than the second, the carry flag would have been cleared, so BCC (Branch if Carry Clear) will test for that possibility. If the first number is higher than the second, BCS (Branch if Carry Set) will be true. You can therefore branch with BEQ for =, BCC for <, and BCS for >. Just keep in mind which number you are considering the *first* and which the *second* in this test.

Double-Byte Addition

CLC ADC and SEC SBC will add and subtract one-byte numbers. To add two-byte numbers, use:

> (Assume that the numbers you want to add are located in addresses \$0605,0606 and \$0700,0701. The ML program segment performing the addition is located at \$5000.)

5000 CLC; always do this before any addition
5001 LDA \$0605
5004 ADC \$0700
5007 STA \$0605; the result will be left in \$0605,0606
500A LDA \$0606
500D ADC \$0701
5010 STA \$0606

It's not necessary to put the result on top of the number in \$0605,0606—you can put it anywhere. But you'll often be adding a particular value to another and not needing the original any longer—adding ten points to a score for every blasted alien is an example. If this were the case, following the logic of the routine above, you would have a 10 in \$0701, 0702:

0701 0A; the 10 points you get for hitting an alien
0702 00

You'd want that 10 to remain undisturbed throughout the game. The score, however, keeps changing during the game and, held in \$0605,0606, it can be covered over, replaced with each addition.

Double-Byte Subtraction

This is quite similar to double-byte addition. Since subtracting one number from another is also a comparison of those two numbers, you could combine subtraction with the double-byte comparison routine above (using ORA). In any event, this is the way to subtract double-byte numbers. Be sure to keep straight which number is being subtracted from the other. We'll call the number *being subtracted* the *second number.*

> (Assume that the number you want to subtract [the "second number"] is located in addresses \$0700,0701, and the number it is being subtracted from [the "first number"] is held in \$0605,0606. The result will be left in \$0605,0606. The ML program segment performing the subtraction is located at \$5000.)

```
5000 SEC; always do this before any subtraction
5001 LDA $0605; low byte of first number
5004 SBC $0700; low byte of second number
5007 STA $0605; the result will be left in $0605,0606
500A LDA $0606; high byte of first number
500D SBC $0701; high byte of second number
5010 STA $0606; high byte of final result
```

Multi-Byte Addition and Subtraction

Using the methods for adding and subtracting illustrated above, you can manipulate larger numbers than can be held within two bytes (65535 is the largest possible two-byte integer). Here's how to subtract one four-byte-long number from another. The locations and conditions are the same as for the two-byte subtraction example above, except the "first number" (the *minuend*) is held in the four-byte chain, $0605,0606,0607,0608, and the "second number" (the *subtrahend,* the number being subtracted from the first number) is in $0700,0701,0702,0703.

Also observe that the most significant byte is held in $0703 and $0608. We'll use the Y Register for Indirect Y addressing, use four bytes in zero page as pointers to the two numbers, and use the X Register as a counter to make sure that all four bytes are dealt with. This means that X must be loaded with the length of the chains we're subtracting—in this case, 4.

```
5000 LDX #4; length of the byte chains
5002 LDY #0, set Y
5004 SEC; always before subtraction
5005 LOOP LDA (FIRST),Y
5007 SBC (SECOND),Y
5009 STA (FIRST),Y; the answer will be left in $0605-0608.
500B INY; raise index to chains
500C DEX; lower counter
5010 BNE LOOP; haven't yet done all four bytes
```

Before this will work, the pointers in zero page must have been set up to allow the Indirect Y addressing. This is one way to do it:

```
2000 FIRST = $FB; define zero page pointers at $FB and $FD
2000 SECOND = $FD
2000 SETUP LDA #5; set up pointer to $0605
2002 STA FIRST
2004 LDA #6
2006 STA FIRST+1
2008 LDA #0; set up pointer to $0700
200A STA SECOND
200C LDA #7
200E STA SECOND+1
```

Multiplication

× 2

ASL (no argument used, "Accumulator addressing mode") will multiply the number in the Accumulator by 2.

× 3

(To multiply by 3, use a temporary variable byte we'll call TEMP.)

```
5000 STA TEMP; put the number into the variable
5003 ASL; multiply it by 2
5004 ADC TEMP; (X * 2 + X = X * 3) the answer is in A.
```

× 4

(To multiply by 4, just ASL twice.)

```
5000 ASL; * 2
5001 ASL; * 2 again
```

× 4 (two byte)

(To multiply a two-byte integer by 4, use a two-byte variable we'll call TEMP and TEMP+1.)

```
5000 ASL TEMP; multiply the low byte by 2
5003 ROL TEMP+1; moving any carry into the high byte
5006 ASL TEMP; multiply the low byte by 2 again
5009 ROL TEMP+1; again acknowledge any carry.
```

× 10

(To multiply a two-byte integer by 10, use an additional two-byte variable we'll call STORE.)

```
5000; first put the number into STORE for safekeeping
5000 LDA TEMP:STA STORE:LDA TEMP+1:STA STORE+1
500C; then multiply it by 4
500C ASL TEMP; multiply the low byte by 2
500F ROL TEMP+1; moving any carry into the high byte
5012 ASL TEMP; multiply the low byte by 2 again
5015 ROL TEMP+1; again acknowledge any carry.
5018; then add the original, resulting in X * 5
5018 LDA STORE
501B ADC TEMP
501E STA TEMP
5021 LDA STORE+1
501D ADC TEMP+1
5024 STA TEMP+1
5027; then just multiply by 2 since (5 * 2 = 10)
5027 ASL TEMP
502A ROL TEMP+1
```

× ?

(To multiply a two-byte integer by other odd values, just use a similar combination of addition and multiplication which results in the correct amount of multiplication.)

× 100

(To multiply a two-byte integer by 100, just go through the above subroutine twice.)

× 256

(To multiply a one-byte integer by 256, just transform it into a two-byte integer.)

```
5000 LDA TEMP
5003 STA TEMP+1
5006 LDA #0
5008 STA TEMP
```

Division

÷ 2

LSR (no argument used, "Accumulator addressing mode") will divide the number in the Accumulator by 2.

÷ 4

(To divide by 4, just LSR twice.)

5000 LSR; / 2
5001 LSR; / 2 again

÷ 4 (two byte)

(To divide a two-byte integer, called TEMP, by 2)

5000 LSR TEMP+1; shift high byte right
5001 ROR TEMP; pulling any carry into the low byte

How to Type In BASIC Programs

Some of the programs listed in this book are written in BASIC and contain special control characters (cursor control, color keys, inverse video, etc.). To make it easy to tell exactly what to type when entering one of these programs into your computer, we have established the following listing conventions. There is a separate key for each computer. Refer to the appropriate tables when you come across an unusual symbol in a program listing. If you are unsure how to actually enter a control character, consult your computer's manuals.

Atari

Characters in inverse video will appear like: INVERSE VIDEO
Enter these characters with the Atari logo key, ().

When you see	Type	See	
{CLEAR}	ESC SHIFT <	↖	Clear Screen
{UP}	ESC CTRL -	↑	Cursor Up
{DOWN}	ESC CTRL =	↓	Cursor Down
{LEFT}	ESC CTRL +	←	Cursor Left
{RIGHT}	ESC CTRL *	→	Cursor Right
{BACK S}	ESC DELETE	◂	Backspace
{DELETE}	ESC CTRL DELETE		Delete Character
{INSERT}	ESC CTRL INSERT		Insert Character
{DEL LINE}	ESC SHIFT DELETE		Delete Line
{INS LINE}	ESC SHIFT INSERT		Insert Line
{TAB}	ESC TAB	▸	TAB key
{CLR TAB}	ESC CTRL TAB		Clear TAB
{SET TAB}	ESC SHIFT TAB		Set TAB stop
{BELL}	ESC CTRL 2		Ring Buzzer
{ESC}	ESC ESC		ESCape key

Graphics characters, such as CTRL-T, the ball character ● will appear as the "normal" letter enclosed in braces, e.g., {T}.

A series of identical control characters, such as 10 spaces, 3 cursor-lefts, or 20 CTRL-Rs, will appear as {10 SPACES}, {3 LEFT}, {20 R}, etc. If the character in braces is in inverse video, that character or characters should be entered with the Atari logo key. For example, {5 } means to enter five inverse-video CTRL-Us.

Commodore 64, VIC, and PET

Program listings will contain words within braces which spell out any special characters: {DOWN} would mean to press the cursor down key. {5 SPACES} would mean to press the space bar five times.

To indicate that a key should be *shifted* (hold down the SHIFT key while pressing the other key), the key would be underlined in our listings. For example, S would mean to type the S key while holding the SHIFT key. If you find an underlined key enclosed in braces (e.g., {10 N}), you should type the key as many times as indicated (in our example, you would enter ten shifted Ns).

If a key is enclosed in special brackets, ⊧< >⫤, you should hold down the *Commodore* key while pressing the key inside the special brackets. (The Commodore key is the key in the lower left corner of the keyboard.) Again, if the key is preceded by a number, you should press the key as many times as indicated.

Rarely, you'll see a solitary letter of the alphabet enclosed in braces. These characters can be entered by holding down the CTRL key while typing the letter in the braces. For example, {A} would indicate that you should press CTRL-A.

About the *quote mode*: You should know that you can move the cursor around the screen with the CRSR keys. Sometimes a programmer will want to move the cursor under program control. That's why you see all the {LEFT}'s, {HOME}'s, and {BLU}'s in our programs. The only way the computer can tell the difference between direct and programmed cursor control is the quote mode.

Once you press the quote (the double quote, SHIFT-2), you are in the quote mode. If you type something and then try to change it by moving the cursor left, you'll only get a bunch of reverse-video lines. These are the symbols for cursor left. The only editing key that isn't programmable is the DEL key; you can still use DEL to back up and edit the line. Once you type another quote, you are out of quote mode.

You also go into quote mode when you INserT spaces into a line. In any case, the easiest way to get out of quote mode is to just press RETURN. You'll then be out of quote mode and you can cursor up to the mistyped line and fix it.

Use the following tables when entering special characters:

When You Read:	Press:	See:	When You Read:	Press:	See:
{CLR}	SHIFT CLR HOME		{GRN}	CTRL 6	
{HOME}	CLR HOME		{BLU}	CTRL 7	
{UP}	SHIFT ↑ CRSR ↓		{YEL}	CTRL 8	
{DOWN}	↑ CRSR ↓		{F1}	f1	
{LEFT}	SHIFT ← CRSR →		{F2}	f2	
{RIGHT}	← CRSR →		{F3}	f3	
{RVS}	CTRL 9		{F4}	f4	
{OFF}	CTRL 0		{F5}	f5	
{BLK}	CTRL 1		{F6}	f6	
{WHT}	CTRL 2		{F7}	f7	
{RED}	CTRL 3		{F8}	f8	
{CYN}	CTRL 4		←	←	
{PUR}	CTRL 5		↑	SHIFT ↑	

Index

To order your copy of the LADS Disk call our toll-free US order line: 1-800-334-0868 (in NC call 919-275-9809) or send your prepaid order to:

LADS Disk
COMPUTE! Publications
P.O. Box 5406
Greensboro, NC 27403

Please specify whether you want the Apple, Atari or Commodore LADS. All orders must be prepaid (check, charge, or money order). NC residents add 4.5% sales tax.

Send ____ copies of the LADS Disk at $12.95 per copy for (check one) ☐ Apple ☐ Atari ☐ Commodore

Subtotal $__________

Shipping & Handling: $1.00/disk* $__________

Sales tax (if applicable) $__________

Total payment enclosed $__________

*Outside US and Canada, add $3.00 per disk for shipping and handling. All payments must be in US funds.

☐ Payment enclosed
Charge ☐ Visa ☐ MasterCard ☐ American Express

Acct. No. ______________________________ Exp. Date ______

Name ____

Address ____

City ____________________ State ________ Zip ________

Please allow 4-5 weeks for delivery.

ERRATA

A Note To VIC-20 Users:

To insure reliable assembly with the LADS assembler on the VIC, leave the .S (print to screen) pseudo-op active at all times.

This is the companion volume to the best seller, *Machine Language for Beginners*, about which the critics have said:

> *"Understandable"*—The New York Times
>
> *"Presents the machine language novice with a very good tutorial in simple, understandable terms."*—Antic
>
> *"I highly recommend* Machine Language for Beginners *as your first introduction to the world of machine language."*—Commodore Power/Play
>
> *"This is an excellent book for anyone considering learning machine language programming. It is well written and easy to follow, and everything is presented in a logical and orderly fashion."*—RUN
>
> *"Highly recommended . . . usable, understandable . . . abounds in illustrations, standards and examples."*
> —The Midnight Gazette
>
> *"It lives up to its title. I have about six books on this subject, and this is by far the most readable of the group."*—VICtims Newsletter
>
> *"The best book on introducing 6502 code available."—Ian Chadwick, author of* Mapping the Atari

The Second Book of Machine Language picks up where *Machine Language for Beginners* left off. This new book contains one of the most powerful machine language assemblers currently available. The LADS assembler is a full-featured, label-based, programming language which can greatly assist you in writing machine language programs quickly and easily.

You work in an environment with which you're already familiar: BASIC. You can use line numbers, multiple statements on a line, named variables and subroutines, remarks, error messages, and various programmers' aids like automatic line numbering, search and replace, etc.

But the book is more than a sophisticated program. It's also a clear, detailed tutorial on how large, complex machine language programs can be constructed out of manageable subprograms. Using LADS as the example, each instruction is explained, each subroutine is examined. Many sophisticated machine language techniques are illustrated and thoroughly explained—everything from data base management to communication with printers and disk drives.

There are powerful computer languages and there is good documentation, but rarely has a sophisticated language been so completely documented as it is in this book. When you finish with this book, you'll not only have a deeper understanding of machine language—you'll also have one of the most powerful machine language assemblers available.

ISBN 0-942386-5301

[illegible]nodore 64, Apple (II, II+, IIe, and IIc, DOS 3.3), VIC-20 (8K RAM expansion required), Atari (inc[illegible]ing XL, 40K minimum), and PET/CBM (Upgrade and 4.0 BASIC).