# Assembly Cookbook for the Apple® II/IIe

Don Lancaster

Assembly Cookbook for the Apple II/IIe

Don Lancaster heads *Synergetics,* a new-age prototyping and consulting firm involved in micro applications and electronic design. He is the well-known author of the classic *CMOS* and *TTL Cookbooks.*

He is one of the microcomputer pioneers, having introduced the first hobbyist integrated-circuit projects, the first sanely priced digital electronics modules, the first low-cost TVT-1 video display terminal, the first personal computing keyboards, and lots more. Don's numerous books and articles on personal computing and electronic applications have set new standards as understandable, useful, and exciting technical writing.

Don's other interests include ecological studies, firefighting, cave exploration, bicycling, and tinaja questing.

The Don Lancaster Library

Assembly Cookbook
for the Apple II/IIe

by

Don Lancaster

Contents

part I
Some Theory

part II
The Ripoff Modules

Appendixes

0

WHY YOU GOTTA LEARN ASSEMBLY LANGUAGE

Check into *Softalk* magazine's listing of the "top thirty" programs for your Apple II or IIe, and you'll find that thirty out of thirty of this month's winners usually involve machine language programs or support modules, written by authors who use assemblers and who make use of assembly language programming skills.

And, last month's top thirty were also swept by machine language, thirty to zip. And next month's listings probably will be the same. Somehow, thirty to zero seems statistically significant. There's got to be a message there.

Yep.

So, on the basis of what is now happening in the real world, you can easily conclude that . . .

> **The only little thing wrong with BASIC or Pascal is that it is categorically impossible to write a decent Apple II or IIe program with either of them!**

Naturally, things get even worse if you try to work in some specialty language, such as FORTH, PILOT, LOGO, or whatever, since you now have an even smaller user and interest base and thus an even more miniscule market.

What would happen if, through fancy packaging, heavy promotion,

or outright lies, a BASIC or a Pascal program somehow happened to blunder into the top thirty some month?

One of three things . . .

—maybe—

1. Word will quickly get out over the bulletin board systems and club grapevines over how gross a ripoff the program is, and the program will ignominiously bomb out of sight.

—or—

2. A competitor will recognize a germ or two of an undeveloped idea in the program and come up with a winning machine language replacement that does much more much faster and much better, thus running away with all the marbles.

—or, hopefully—

3. The program author will see the blatant stupidity of his ways and will rework the program into a decent, useful, and popular machine language version.

The marketplace has spoken, and its message is overwhelming . . .

> **If you want to write a best-selling or money-making program for the Apple II or IIe, the program *must* run in machine language.**

OK, so it's obvious that all the winning Apple II programs run in machine language. But, *why* is this so? What makes machine language so great? How does machine language differ from the so-called "higher level" languages? What is machine language all about?

Here are a few of the more obvious advantages of machine language . . .

MACHINE LANGUAGE IS—
Fast
Compact
Innovative
Economical
Flexible
Secure
User Friendly
Challenging
Profitable

That's a pretty long list and a lot of heavy claims. Let's look at a few of the big advantages of machine language one by one . . .

Speed

It takes from two to six millionths of a second, or *microseconds,* to store some value using Apple's 6502 machine language. Switch to interpreted Integer BASIC or Applesoft, and similar tasks take as much as two to six thousandths of a second, or *milliseconds.* This is slower by a factor of one thousand.

The reason for the 1000:1 speed difference between interpreted "high level" languages and machine language is that there are bunches of housekeeping and overhead involved in deciding which tasks have to be done in what order, and in keeping things as programmer friendly as possible.

Now, at first glance, speed doesn't seem like too big a deal. But speed is crucial in many programs. Let's look at three examples.

For instance, a word processor program that inserts characters slower than you can type is a total disaster, for one or more characters can get dropped. Even if it doesn't drop characters, a word processor that gets behind displaying stuff on the screen gets to be very infuriating and annoying. So, word processing is one area where machine language programs are an absolute must, because of the needed speed.

Business sorts and searches are another area where the speed of machine language makes a dramatic difference. Several thousand items sorted in interpreted BASIC using a bubble sort might take a few hours. Go to a quicksort under machine language, and the same job takes a few seconds at most. Thus, any business program that involves sorts and searches of any type is a prime candidate for machine language.

Finally, there is any program that uses animation. Interpreted BASIC is way too slow and far too clumsy to do anything useful in the way of screen motion, game responses, video art, and stuff like this. Thus, *all* challenging or interesting games need machine language to keep them that way.

But, you may ask, what about compilers? Aren't there a bunch of very expensive programs available that will compile BASIC listings into fast-running machine language programs?

Sure there are.

Most compiled code usually runs faster than interpreted code. But, when you find the real-world speedup you get and compare it to the same program done in machine language by a skilled author, it is still no contest . . .

> **Most programs compiled from a "higher level" language will run far slower, and will perform far more poorly, than the same task done in a machine language program written by a knowing author.**

Some specifics. If you run exactly the worst-case benchmark program on one of today's highly promoted compiler programs, you get a blinding speedup of *8 percent,* compared to just using plain old interpreted BASIC. Which means that a task that took two hours and fifty-five minutes can now be whipped through in a mere two hours and forty-two minutes instead.

Golly gee, Mr. Science.

Actually, most compilers available today will in fact speed up interpreted programs by a factor of two to five. This is certainly a noticeable difference and is certainly a very useful speedup. But it is nothing compared to what experienced machine language authors can do when they attack the same task.

A compiler program has to make certain assumptions so that it can work with all possible types of program input. Machine language authors, on the other hand, are free to optimize their one program to do whatever has to be done, as fast, as conveniently, and as compactly as possible. This is the reason why you can *always* beat compiled code if you are at all into machine language.

Another severe limitation of Applesoft compilers is that they still end up using Applesoft subroutines. These subroutines may be just plain wrong (such as RND), or else may be excruciatingly slow (such as HPLOT). Hassles like these are easily gotten around by programming directly in machine language.

Some machine language programs are faster than others. Most often, you end up trading off speed against code length, programming time, and performance features.

One way to maximize speed of a machine language program is to use *brute-force coding,* in which every instruction does its thing in the minimum possible time, using the fastest possible addressing modes. Another speed trick is called *table lookup,* where you look up an answer in a table, rather than calculating it. One place where table lookup dramatically speeds things up is in the Apple II HIRES graphics routines where you are trying to find the address of a display line. Similar table lookups very much quicken trig calculations, multiplications, and stuff like this.

So, our first big advantage of machine language is that it is ridiculously faster than an interpreted high level language, and much faster than a compiled high level language.

Size

A controller program for a dumb traffic light can be written in machine language using only a few dozen bytes of code. The same thing done with BASIC statements takes a few hundred bytes of code, not counting the few thousand bytes of code needed for the BASIC interpreter. So, machine language programs often can take up far less memory space than BASIC programs do.

Now, saving a few bytes of code out of a 64K or 128K address space may seem like no big deal. And, it is often very poor practice to spend lots of time to save a few bytes of code, particularly if the code gets sneaky or hard to understand in the process.

But, save a few dozen bytes, and you can add fancy sound to your program. Save a few thousand bytes more, and you can add HIRES graphics or even speech. Any time you can shorten code, you can make room for more performance and more features, by using up the new space you created. Save bunches of code, and you can now do stuff on a micro that the dino people would swear was impossible.

Three of the ways machine language programs can shorten code include using *loops* that use the same code over and over again, using *subroutines* that let the same code be reached from different places in

a program, and using *reentrant* code that calls itself as often as needed. While these code shortening ideas are also usable in BASIC, the space saving results are often much more impressive when done in machine language.

Machine language programs also let you put your files and any other data that go with the program into its most compact form. For instance, eight different flags can be stuffed into a single code word in machine language, while BASIC normally would need several bytes for each individual flag.

Which brings us to another nasty habit compilers have.

Compilers almost always make an interpreted BASIC program *longer* so that the supposedly "faster" compiled code takes up even more room in memory than the interpreted version did. The reason for this is that the compiler must take each BASIC statement at face value, when and as it comes up. The compiler then must exactly follow the *form* and *structure* of the original interpreted BASIC code. Thus, what starts out as unnecessarily long interpreted code gets even longer when you compile it.

Not to mention the additional interpretive code and run time package that is also usually needed.

A machine language programmer, on the other hand, does not have to take each and every BASIC statement as it comes up. Instead, he will write a totally new machine language program that, given the same inputs, provides the same or better outputs than the BASIC program did. This is done by making the new machine language program have the same *function* that the BASIC one did, but completely ignoring the dumb *structure* that seems to come with the BASIC territory.

The net result of all this is that a creative machine language programmer can often take most BASIC programs and rewrite them so they are actually shorter. As a typical example, compare your so-so adventure written in BASIC against the mind blowers written in machine. When it comes to long files, elaborate responses, and big data bases, there is no way that BASIC can compete with a machine language program, either for size or speed.

Let's check into another file-shortening example, to see other ways that machine language can shorten code. The usual way a higher level language handles words and messages is in ASCII code. But studies have shown that ASCII code is only 25 percent efficient in storing most English text. Which means that you can, in theory, stuff *four times* as many words or statements into your Apple as you thought you could with ASCII.

You do this by using some *text compaction* scheme that uses non-standard code manipulated by machine language instructions. For instance, in *Zork*, three ASCII characters are stuffed into two bytes of code. This gives you an extra 50 percent of room on your diskettes or in your Apple. In the *Collossial Cave* adventure version by Adventure International, unique codings are set aside for *pairs* of letters, giving you up to 100 percent more text in the same space. This means that this *entire* classic adventure text now fits inside the Apple, without needing *any* repeated disk access.

Dictionary programs use similar compaction stunts to minimize code length. If the words are in alphabetical order, you can play another compaction game by starting with a number that tells you how many of the beginning letters stay the same, and by using

another coding scheme to add standard endings (-s, -ing, -ed, -ly, etc. . . .) to the previous word.

The bottom line is that machine language programs can shorten code enough that you can add many new features to an existing program, can put more information in the machine at once, or can cram more data onto a single diskette.

Innovation—Finding the Limits

One really big advantage to machine language on the Apple II or IIe is that it pushes the limits of the machine to the wall. We now can do things that seemed impossible only a short while ago. This is done by discovering new, obscure, and mind-blowing ways to handle features using machine language code.

Some ferinstances.

With BASIC, you can get only one obnoxious beep out of the Apple's on-board speaker. Play around with PEEKs and POKEs, and you can get a few more pleasant buzzes and low-frequency notes. This is almost enough to change a fifth rate program into a fourth rate one.

Now, add a short machine language program, and you can play any tone of any duration. But, that's old hat. The big thing today is called *duty cycling*. With duty cycling done from a fairly fancy machine language driver, you can easily sound the on-board speaker at variable volume, with several notes at once, or even do speech with surprisingly good quality.

All this through the magic of machine language, written by an author who uses assemblers and who posesses assembly language programming skills.

The Apple II colors are another example. The HIRES subs in BASIC only give you 16 LORES colors and a paltry 6 HIRES colors. But, go to machine language, and you end up with at least 121 LORES colors and at least 191 HIRES ones on older Apples. The Apple IIe offers countless more.

And that's today. Even more colors are likely when the machine language freaks really get into action.

Another place where limits are pushed by machine language is in animation and HIRES plotting. You can clear the HIRES screen seven times faster than was thought possible, by going to innovative code. You can plot screen locations much faster today through the magic of table lookup and brute-force coding. Classic cell animation is even possible.

Disk drive innovations are yet another example. Change the code and you can load and dump diskettes several times faster than you could before. You can also store HIRES and LORES pictures in many fewer sectors than was previously thought possible. Again, it is all done by creative use of machine language programs that are pushing the limits of the Apple.

A largely unexplored area of the Apple II involves *exact field sync*, where an exact and jitter-free lock is done to the screen. This lets you mix and match text, LORES, and HIRES on the screen, do gray scale, precision light pens, gentle scrolls, touch screens, flawless animation, and much more.

All this before the magic of the new cucumber cool 65C02 chips,

which can allow a mind-boggling animation speedup of *fifty times,* compared to what the best of today's machine language programmers are using. But that's another story for another time.

And, exciting as the pushed limits are, we are nowhere near the ultimate . . .

> **Today's machine language programs are nowhere near pushing the known limits of the Apple II's hardware.**

And, of course . . .

> **The *known* limits of the Apple II hardware are nowhere near the *real* limits of the Apple II hardware.**

What haven't we fully explored with the Apple II yet? How about gray scale? Anti-aliasing? Three-D graphics displays? "Picture processing" for plotters that is just as fast and convenient as "word processing"? Using the Apple as an oscilloscope? A voltmeter? Multi-Apple games, where each combatant works his own machine in real time? Scan length coded video? That 50X animation speedup? Networking?

And the list goes on for thousands more. If it can be done at all, chances are an Apple can help you do it, one way or another.

Getting Rid of Fancy Hardware

Machine language is often fast enough and versatile enough to let you get rid of fancy add-on hardware, or else let you dramatically simplify and cut down on needed hardware. This is why we say that machine language is economical.

For instance, without machine language drivers on older Apples, you are stuck either with a 40-character screen line, or else have to go to a very expensive 80-column card board. But with the right drivers, you can display 40, 70, 80 or even 120 characters on the screen of an *unmodified* Apple II with no plug-in hardware. This is done by going to the HIRES screen and by using more compact fonts. You can also have many different fonts this way, upper or lower case, in any size and any language you like.

As a second example of saving big bucks with machine language, one usual way to control the world with an Apple II involves a BSR controller plug-in card, again full of expensive hardware. But you can replace all this fancy hardware with nothing but some machine language code and a cheap, old, ultrasonic burglar alarm transducer.

As yet a final example, by going to the Vaporlock exact field sync, machine language software can replace all the custom counters needed for a precision light pen or for a touch screen. With zero hardware modifications.

In each of these examples, the machine language code is fast enough that it can directly synthesize what used to be done with fancy add-on hardware.

So, our fourth big plus of machine language is that it can eliminate, minimize, or otherwise improve add-on hardware at very low cost.

Other Advantages

Those are the big four advantages of machine language. Speed, program size, innovation, and economy. Let's look at some more advantages.

Machine language code is very flexible. Have you ever seen Kliban's cartoon "Anything goes in Heaven," where a bunch of people are floating around on clouds doing things that range from just plain weird to downright obscene?

Well, anything goes in machine language as well. Put the program any place you want to. Make it as long or as short as you want. "What do you mean I can't input commas?" Input what you like, when you like, how you like. Change the program anyway you want to, anytime you want to. That's what flexibility is all about.

Machine language offers solace for the security freak.

I'm not very much into program protection myself, since all my programs are unlocked, include full source code, and are fully documented. I, like practically every other advanced Apple freak, fiendishly enjoy tearing apart all "protected" programs the instant they become available, because of the great sport, humor, learning, and entertainment value that the copy protection mafia freely gives us.

And surprise, surprise. Check the *Softalk* score sheets, and you'll find that unlocked programs are consistently outselling locked ones, and are steadily moving *up* in the ratings and in total sales. Which means that an un-displeased and un-inconvenienced buyer in the hand is worth two bootleg copies in the bush, any day.

Time spent "protecting" software is time blown. Why not put the effort into improving documentation, adding new features, becoming more user friendly, or doing more thorough testing instead?

But, anyway, if you are naive enough or arrogant enough to want to protect your program, there are lots of opportunities for you to do so in machine language. For openers, probably 98 percent of today's Apple II owners do not know how to open and view a machine language program. Not only are you free to bury your initials somewhere in the code, but you could hide a seven-generation genealogical pedigree inside as well. How's that for proof of ownership? And, the very nature of creative machine language programming that aims to maximize speed and minimize memory space, tends to "encrypt" your program. Nuff said on this.

Machine language programs can be made very user friendly. Most higher level languages have been designed from the ground up to be designer friendly instead. BASIC goes out of its way to be easy to learn and easy to program. So, BASIC puts the programmer first and the user last. Instead of making things as easy to program as possible, you are free in machine language to think much more about the ultimate user, and make things as convenient and comfortable as possible for the final user.

Machine language programming is challenging.

Is it ever.

When you become an Apple II machine language programmer, you join an elite group of the doers and shakers of Appledom. The doing doggers. This is where the challenge is, and where you'll find all the action.

And all the nickels.

Finally, there is the bottom line advantage, the sum total of all the others. Because machine language programming is fast, compact, innovative, economical, flexible, secure, and challenging, it is also profitable. Machine language is, as we've seen, the *only* way to grab the brass ring and go with a winning Apple II or IIe program.

Should you want to see more examples of innovative use of Apple II and IIe machine language programs, check into the *Enhancing Your Apple II* series (Sams 21822). And for down-to-earth details on forming your own computer venture, get a copy of *The Incredible Secret Money Machine*.

But, surely there must be disadvantages to machine language programming. If machine is so great, why don't all the rest of the languages just dry up and blow away?

Well, there is also . . .

THE DARK SIDE OF MACHINE LANGUAGE

Here's the bad stuff about machine language . . .

MACHINE LANGUAGE IS ALSO—
Unportable
Tedious
Designer Unfriendly
Multi-Level
Hard to Change
Hard to Teach
Unforgiving
Ego Dependent

What a long list. A machine language program is not portable in that it will only run on one brand of machine, and then only if the one machine happens to be in exactly the right operating mode with exactly the right add-ons for that program. This means you don't just take an Apple II machine language program and stuff it into another computer and have it work.

Should you want to run on a different machine, you have to go to a lot of trouble to rewrite the program. Things get even messier when you cross microprocessor family boundaries. For instance, translating an Apple II program to run on an Atari at least still uses 6502 machine language coding. All you have to do is modify the program to meet all the new locations and all the different use rules. But go from the 6502 to a different microprocessor, and all the addressing modes and op codes will change as well.

A lot of people think this is bad. I don't. If you completely and totally optimize a program to run on a certain machine, then that program absolutely *has* to perform better than any old orphan something wandering around from machine to machine looking for a home.

Machine language involves a lot of tedious dogwork. No doubt about it. Where so-called "higher level" languages go out of their way to be easy to program and easy to use, machine language *does* not.

There are, fortunately, many design aids available that make machine language programming faster, easier, and more convenient. Foremost of these is a good assembler, and that is what the rest of this book is all about.

Machine language is very designer unfriendly. It does not hold your hand. A minimum of three years of effort is needed to get to the point where you can see what commitment you really have to make to become a really great machine language programmer.

Machine language needs multilevel skills. The average machine language program consists of three kinds of code. These are the *elemental subroutines* that do all the gut work, the *working code* that manages the elemental subs, and finally, the *high level supervisory code* that holds everything together. In a "higher level" language, the interpreter or compiler handles all the elemental subs and much of the working code for you, "free" of charge. Different skills and different thought processes are involved in working at these three levels.

This disadvantage is certainly worth shouting over . . .

> **THERE IS NO SUCH THING AS A
> "SMALL" CHANGE IN A MACHINE
> LANGUAGE PROGRAM!**

Thus, any change at all in a machine language program is likely to cause all sorts of new problems. You don't simply tack on new features as you need them, or stuff in any old code any old place. This just isn't done.

Actually, shoving any old code any old place is done all the time, by just about everybody. It just doesn't work, that's all.

Machine language programming is something that must be learned. There is no way for someone else to "teach" you machine language programming. Further, the skills involved in becoming a good machine language programmer tend to make you a lousy teacher, and vice versa.

Machine language is unforgiving, in that any change in any byte in the program, or any change in starting point, or any change of user configuration, will bomb the program and plow the works.

Some people claim that machine language code is hard to maintain. But it is equally easy to write a Pascal program that is totally unfixable and undecipherable as it is to write a cleanly self-documenting machine language program. The crucial difference is that machine language gently urges you to think about maintainability, while Pascal shoves this down your throat. Sideways.

Finally, machine language is highly ego-dependent. Your personality determines the type and quality of machine language programs you write. Many people do not have, and never will get, the discipline and sense of order needed to write decent machine language programs. So, machine language programming is not for everyone.

It is only for those few of you who genuinely want to profit from and enjoy your Apple II or IIe.

That's a pretty long list of disadvantages, and it should be enough to scare any sane individual away from machine language. Except for this little fact . . .

> NONE of the disadvantages of machine
> language matter in the least, because
> there is NO OTHER ALTERNATIVE to
> machine language when it comes to
> writing winning programs for your Apple
> II or IIe.

Or to rework the tired joke about the guy who slaved away all his life in Florida and then retired to New Jersey . . .

> Have you ever seen a machine language
> program that was *improved* by rewriting
> it in BASIC or Pascal?

Not that it won't happen. It just isn't very likely, that's all.

GETTING STARTED

Here is how I would have you become a decent machine language programmer. First, you should write, *hand-code,* test, and debug several hundred lines directly in machine language, *without* the use of *any* assembler at all. The reason for this is that . . .

> Before you can learn to program in
> assembly language, you *must* learn how
> to program in machine language!

So many assembler books and courses omit this essential first step! An assembler is simply too powerful a tool to start off with. You must first know what op codes are and how they are used. You must thoroughly understand addressing modes and the different ways you show which addressing mode is in use.

There is a series of nine *discovery modules* in Volume II of *Don Lancaster's Micro Cookbook* (Sams 21829) that will take you step by step through most of the op codes of the Apple's 6502 microprocessor, without using any assembly aids.

After you have done your homework and can tell the difference between a page zero and an immediate addressing command, and after you know whether "page zero, indexed by Y," is a legal command on the Apple, then, and only then, should you move up to a miniassembler, such as the excellent one in Apple's new BUGBYTER.

Then you run another few hundred lines of code through a miniassembler to understand what an assembler can do for you.

Finally, you go on to a full blown assembler and learn assembly language programming. But, you should do this only *after* you have the fundamentals under your belt.

The few tedious "front-end" hours spent doing everything "the hard way" will be more than made up in the speed and *convenience* with which you pick up assembler skills later.

This starting by hand, going to a miniassembler, and only then step-

ping up to a full assembler is the way I would have you become a winning assembly language programmer.

Most people, though, will probably try . . .

SNEAKING UP ON REALITY

For a SIXTH rate program—
   Write it in Pascal.

For a FIFTH rate program—
   Write it in BASIC.

For a FOURTH rate program—
   Write it in BASIC, but use a few
   PEEKs and POKEs.

For a THIRD rate program—
   Write it in BASIC, but use the CALL
   command to link a few short
   machine language code segments.

For a SECOND rate program—
   Write it in BASIC, but use the "&"
   command to link several longer
   blocks of machine language code.

For a FIRST rate program—
   Do the whole thing in machine
   language like you should have done
   in the first place.

The trouble with the "sneaking-up" method is that it takes you forever to see how bad "higher level" languages really are, and you spend all your time goofing around with second-rate code. But, the "sneaking-up" method does eliminate some of the cultural shock of starting straight into machine language programming from scratch.

Instead, start with and stay in machine language.

A PLAN

This book is intended to show you what an assembler is and how to use one to write profitable and truly great Apple II or IIe machine language programs. You will find the book in two halves. The first half is the "theory" part that tells us all about what assemblers are and how to use them. The second half is the "practice" part that will lead you step by step through some practical ripoff modules of working assembly language code. Code that is unlocked, ready to go, wide open, and easily adapted to your own uses.

We start in chapter one by finding out what an assembler is and what *it does*. We then check into the popular assemblers available today, along with a list of the essential tools for assembler programming, some magazines, and other resources.

Our examples will use Apple's own newly overhauled and upgraded EDASM macroassembler, first because it is the one I use,

and secondly because it is the de-facto standard for assembling Apple II machine language programs. Many of the weakest features of EDASM get eliminated in one swell foop simply by using *Apple Writer IIe* instead of the original EDASM editor, and using the magic of WPL to help along your macros.

At any rate, most of what happens here will apply to any assembler of your choosing. We will provide source code on the companion diskette for either EDASM or the S-C Assembler formats. Just be sure to tell us which one you want. Either of these versions should be translatable to the assembler of your choice.

Most Apple-based assemblers come in two parts. One part puts together the story of what is to be done, while the second part takes the story and converts it into working machine language code. Putting together the story is called *editing,* while creating the machine language code is called *assembling.* The story or script is more properly called the *source code,* while the final program or module is usually called the *object code.*

Source code details are covered in chapter two, where we look into source code lines, fields, labels, op codes, operands, and comments, finding out just what all these are and how they are used. The structure of your source code is outlined in chapter three, where we find the 16 essential parts to an assembly language program, and how to use them. We also find out here exactly why structure of any kind is inherently evil and why structure must be avoided at all costs.

Today there are two good ways to write source code. The "old way" uses the editor provided in the assembler package. We'll cover the old way in depth in chapter four.

The "new way" uses the power of a modern word processor to do your source code entry and editing, and has bunches of potent advantages. Not the least of which is that creating and editing source code is lots more fun with a word processor, and that you can instantly upgrade a lousy editor/assembler into a super-powerful one. Drag a supervisory language such as WPL along for the ride, and you can do incredible macro-style things that otherwise would be unavailable. Chapter five tells all.

At long last, in chapter six, we get around to actually assembling source code into working object code. Here we also check into error messages, debugging techniques, and things like this. And that just about rounds out the theory half of the book.

The practice half includes nine ready-to-go ripoff modules that show you examples of some of the really essential stuff involved in Apple programming. I've tried to concentrate on the things that are really needed and really get used, such as a decent random number generator, a state-of-the-art string imbedder, an option picker, a time delay animator, two approaches to sound effects, a classic text handler, a rearranging shuffler, and an empty shell source code builder. I have tried to keep the programs and modules general enough and simple enough that they will run on most any brand or version of Apple or Apple knockoff.

A stuffed-full and double-sided companion diskette is available with all the source and object code used in the book. Source code is provided in your choice of EDASM or S-C Assembler formats. Either way can be used as is, or else easily adapted to most any assembler of your

choice. Naturally, this companion diskette is fully unlocked, easily copied, and bargain priced.

No royalty or license is needed to use any of the ripoff modules in your own commercial programs, so long as you give credit and otherwise play fair. You can order this diskette directly from me by using the order card in the back of this book. A feedback and update card is also included. An aggressive and well-supported voice hotline service is provided free with your diskette order.

By the way, I'd like to do a really advanced sequel to this book that would cover such things as the new 65C02's with their literally *millions* of new op codes and addressing modes just waiting for your use, review some really old stuff like the Sweet Sixteen and the old floating point routines, check into Apple organization and memory maps more, look into the IIe's fantastic new opportunities, do many more ripoff modules, and lots of extra stuff like this. Be sure to use the response card to tell me exactly what you want to see, and which new ripoff modules should be included.

But, getting back to here and now, don't expect this book to teach you assembly programming, because assembly programming cannot be taught. Assembly of machine language code has to be learned through great heaping bunches of hands-on experience and lots of practice. Careful study of other programs is also an absolute must. Hopefully, you can use this book as a guide to show you the way through your own learning process.

Oh yeah. It is disclaimer time again. Apple II is a registered trademark of some obscure outfit out in California. All of the usual names like Atari, Zork, Scott Adam's, VisiCalc, etc., are registered trademarks of whoever. Special thanks to Bob Sander-Cedarlof of S-C Software for his thoughtful proofing comments.

As usual, everything here is pretty much my own doing, done without Apple's knowledge or consent. Which, of course, makes it even better.

Don Lancaster
Fall 1983

This book is dedicated to the secret of the red wall.

May there always be one more.

part I
SOME THEORY

1

WHAT IS AN ASSEMBLER?

Virtually all the winning and truly great Apple II or IIe programs written today run in *machine language . . .*

> **MACHINE LANGUAGE—**
>
> **The detailed, "ones-and-zeros," gut-level commands a microcomputer must have to do anything.**

For instance, if the Apple's data bus is presented with the binary pattern 1010 1001 and then with 1011 0111, the 6502 microprocessor will fill its accumulator with the value hexadecimal $B7, equal to decimal 183. This is done in the immediate addressing mode, as a two-byte instruction.

If the previous paragraph looks like so much gibberish, you are not nearly ready to even think about reading this book.

To continue, you must know about and must have used 6502 op codes, and must completely and thoroughly understand addressing modes and hexadecimal notation. Memory maps, working registers, and address space must be second nature to you. You must also have already *handwritten* and hand debugged several of your own machine language programs.

Once again, this book is about assemblers, and there is NO WAY you should be using assemblers and assembly language until long

after you have handwritten and hand debugged not less than several hundred lines of machine language code.

One place to pick up this machine language background is with the discovery modules in *Don Lancaster's Micro Cookbooks,* Volumes I and II (Sams 21828 and 21829).

At any rate . . .

**If you have not done your hex
homework, can't tell immediate from
page zero addressing, or otherwise
haven't paid your machine language
dues, then . . .**

GO AWAY!

Now that the air is cleared, and the techno-turkeys have left, let's sweep up the worst of the feathers and continue.

The trouble with machine language programs, as you undoubtedly know by now, is that there is lots of tedious dogwork involved in writing them.

It is very hard to insert something new into a hand-coded machine language program, since you have to move everything down on your programming form to make room for new stuff. Removing code creates the opposite problem. Even a common beginner's mistake such as the wrong addressing mode can completely mess up your program. This can easily happen if you have to substitute a 2-byte for a 3-byte instruction, or vice versa.

Calculating relative branches is a royal pain, particularly the forward ones where you don't quite know where you are headed. You must, of course, eat, sleep, and breathe with your 6502 pocket card before you can do any decent machine language programming. You have to know the exact address you are going to jump to, and the exact length of your code, and the exact starting point before the program will work. And you must, of course, do everything in hexadecimal, even if you really want decimal numbers or ASCII characters.

And those are just a few of the hassles. You probably have a lot more pet peeves of your own.

You can automate much of the dogwork involved in machine language programming by going to an *assembler* . . .

ASSEMBLER—

**Any tool that simplifies or automates
machine language programming.**

Most often, an assembler is a program or a program system that you run on your Apple II or IIe that helps you write machine language code. Assemblers make the writing and debugging of machine language code much easier, much faster, and much more fun. Assemblers also make it very easy to change, or *edit,* already existing machine language programs.

Because assembly programs are very powerful tools, there are many new skills involved in learning how to use one. In exchange for this

new learning effort, an assembler will make machine language pro-
gramming much faster and much more fun for you.

The tradeoff is some new effort now in exchange for lots of time
saved and use convenience later.

Assemblers speak a special language that is called, of all things,
assembly language . . .

ASSEMBLY LANGUAGE—

A "higher level" language that both an
assembler and a person wanting to write
machine language programs can use and
understand.

The assembler itself is really one or more machine language pro-
grams set up to interact with you as programmer and the Apple II or
IIe as computer. The assembler goes between you and the machine
and tries to speak to the machine in machine language and to you in
assembly language.

This is roughly similar to an *interpreter* program that can take BASIC
statements understood by a programmer and convert them into
machine language commands understood by the Apple II. An inter-
preter itself is, of course, a machine language program. So is an
assembler.

Thus, you can think of an assembler as a translator that changes
"people language" into "machine language."

Assemblers use *mnemonics* . . .

MNEMONIC—

A group of three or four letters forming a
"word" that both the programmer and
the assembler can understand.

Typical mnemonics would include the command LDX, meaning
"Please load the X register," or ROL A, meaning "Please rotate the
contents of the accumulator to the left through the carry flag." You
should, of course, be already familiar with these mnemonics for 6502
op codes.

Assemblers will often add their own new mnemonics on top of the
ones already used by the 6502. An example would be the mnemonic
ORG, telling the assembler that "Here is the *origin* address where I
would like you to start assembling code." More on these *pseudo-ops*
later.

Mnemonics give us a shorthand way of communicating with an
assembler. We could say "1010 1001 1011 0111" and our 6502
would know what we were talking about. But these ones and zeros
sure get rough on the programmer. We could instead say, "6502,
would you please immediately put the decimal value 183 into your
accumulator?" This is obvious to the programmer, but the 6502's
microprocessor would be very puzzled over this strange gibberish.

Instead, an assembler compromises in its use of mnemonics. The
person uses "LDA #$B7" or "LDA #183" to talk to the assembler,

and the assembler then recognizes and understands this to mean that the machine language coding 1010 1001 1011 0111, or its hex equivalent $A9 $B7, is to go into the final program.

Another very important assembler concept is called a *label* . . .

```
LABEL—

A name put on some value or some
address or some point in a program to be
assembled.
```

Labels are simply names you can put on things. For instance, you could start your program with a label that says START. Other places in your program could *refer* to that label. For instance, to repeat your program over and over again, you could use an assembler command of JMP START as your last line. When the assembler assembles the program, it finds out where START really is and then figures out the right code to get you there. Or maybe you want a forward branch that goes to code you labeled MORE. If you don't use a label, you must know the *exact* address you are branching to, even if you have not gotten there yet. With proper use of labels, a good assembler will automatically figure these things out for you.

Labels also serve as memory joggers and simplify moving programs between machines. For instance, the Apple II's on-board speaker is located at $C030. With a label, you can define, or *equate*, $C030 as SPKR. Every place you see SPKR in a program, you can now remember what it is and what it does.

Another use of labels lets you move your program around in memory by reassembling to a different starting address. If you insert or remove code inside relative branches, those labeled branches will automatically lengthen or shorten during the assembly process. If your branch goes to an absolute address instead, the lengthened or shortened code will bomb, since the branch now goes to the wrong place.

Labels are normally five to seven characters long, and can include numbers or decimal points. Usually you have to start with a letter, and no spaces are allowed. You should try to make all labels as meaningful as possible.

There are lots of sneaky and elegant uses for labels. For instance, you can use the label "C" for a carriage return, or labels of "G1#" and "DQ" to produce a musical note. Labels such as "MSP1" can automate linking of messages and message pointers. You can also do automatic address calculations by combining labels with the upcoming operand arithmetic.

TYPES OF ASSEMBLERS

There are several different types of assemblers, depending on how fancy they are, what they are intended to do, and where they put their final machine language program results.

The simplest is the *miniassembler* . . .

> **MINIASSEMBLER—**
>
> An "automated pocket card" that
> assembles one command at a time
> directly into machine language.

A miniassembler is the smallest and simplest assembler you can get. There is a miniassembler built into the Integer BASIC code of your Apple, starting at $F666. The miniassembler is available either as part of the Integer ROM set, or as code booted onto a language card or into high RAM. The old miniassembler has recently been upgraded and dramatically improved as the BUGBYTER program, available as part of Apple's *Workbench* series.

The system master diskette for Apple IIe will autoboot the miniassembler code for you to power up. To activate it, type INT, followed by a CALL -151, followed by F666G. The same procedure works on the Franklin as well. Or, better yet, BRUN BUGBYTER.

At any rate, all a miniassembler does is let you enter a mnemonic. It then converts that mnemonic to a machine language op code for you. For instance, you tell the miniassembler 0800: LDA #$B7 and the miniassembler whips out its own pocket card, and enters the code into the Apple as 0800: A9 B7. Miniassemblers will automatically calculate relative branches for you, although you often have to make a "guess" on your forward branches, replacing the guess with the real value when you get there.

The use rules for the Apple miniassembler appear in the BUGBYTER manual, and in the usual Apple guides and support books, so we won't repeat them here. But, be absolutely sure you use and thoroughly understand the miniassembler and all of BUGBYTER before you try to tackle anything heavier.

A miniassembler does not allow labels and does not let you write a coded script ahead of time. You simply punch in one mnemonic at a time, and it then changes the mnemonic to an equivalent machine language op code for you. Miniassemblers do not give you a quick and easy way to "open up" code to stuff another command in, or "close up" listings to remove something no longer needed. You usually also must always work in hexadecimal with a miniassembler.

There is no really useful way to put annotation, remarks, or *comments* into a miniassembled program. While you are free to run your printer as you use a miniassembler, there is no really good way to get a well documented hard-copy record of what you are doing.

By *comments,* we mean . . .

> **COMMENTS—**
>
> Remarks or notes added to the
> instructions given an assembler.
>
> Comments are ignored by the assembler,
> but are most useful to people reading
> and using them.

Miniassemblers also assemble code directly into the machine. This means that the code must go into a place in your Apple where it is

expected to run. Trying to assemble code into certain areas such as ROM is futile, and trying to assemble into the stack, the text screen, or much of page zero, will bomb your Apple. So, a miniassembler is normally used to assemble code exactly where it is to run.

The BUGBYTER module is relocatable, so you can move it out of the road of your intended assembly space.

You'll find miniassemblers to be compact and very fast. They are a giant step up from writing your own machine language code by hand, and you always should learn and use a miniassembler before you go on to anything fancier . . .

Before you try to use any fancier assembler, be sure to write and debug not less than several hundred lines of machine language code using a miniassembler or on BUGBYTER.

The greatest use of miniassemblers is to drive home what the assembly process is and how it works. They are also useful for quick-and-dirty or very short assembly jobs. But, since there is no way to make a script of what you want to do, any later changes mean you have to miniassemble the whole job over again. *Worse yet, there's no record of what you did.*

Our next step up leads to a *full assembler* . . .

FULL ASSEMBLER—

An assembler that includes all of the usual features, such as labels, comments, and the ability to work from a script that you can edit and save.

Full assemblers usually consist of a few related programs. One of these lets you write a script, or a series of instructions. You can save this script to disk, edit it, or rework it. A second part of the assembler then converts this script into actual machine language code, and gives you a printed record of the assembly process. Full assemblers use labels and make it easy to insert and remove code at any place and any time. A full assembler is normally all you need to write most machine language programs.

Full assemblers do let you put comments anywhere you like, provide for "pretty printing" for easy readability, and give you a formal printed record.

There is also the *macroassembler* . . .

MACROASSEMBLER—

A full assembler that also is programmable, letting you work with pre-defined modules, and doing other powerful tricks.

A macroassembler will not only accept mnemonics. It will also

accept a pre-defined series of instructions, and then convert those instructions into individual mnemonics.

A *macro* is . . .

MACRO—

A series of instructions or mnemonics that will carry out some fancy "high level" task.

For instance, with a full assembler, you would need a dozen mnemonics to read a text file and print one character at a time. With a macroassembler, you can design a macro that automatically will generate all the needed mnemonics for you. You would name your macro something like PRNTX, and just put the macro into the assembler where you wanted all the details. At the same time, you can "pass variables" through to your macro, such as the name or address where the text message file will start.

A really great macroassembler will even let you use the message *inside* the macro, such as "PRNTX/Hit any key to continue./".

Macros can be instructions inset directly into your source code, can be disk-based modules, or can be WPL routines used with "new way" editing. Which you use depends both on your choice of assembler and your programming style.

Macros are a fun tool for the advanced programmer, and they really make machine language programming fast and more understandable. But macro features are really not essential. If your regular assembler has the ability to insert routines from a disk and can do a limited amount of conditional assembly, you can "fake" many of the macro features without too many hassles.

It is possible to do many macro-like tasks with a supervisory word processor program, such as WPL. WPL is the word processor language that works with *Apple Writer II* or *IIe*. We'll see WPL use examples when we get to the chapter on "new way" editing.

Some assemblers also may give you a method to separate labels that can be used anywhere in the program from labels that can only be used in one small portion of the code. We call these separable names *global* or *local* labels . . .

GLOBAL LABEL—

A label that can be used any place in the entire program.

Global labels can only be defined ONCE in a program.

LOCAL LABEL—

A label that can be used in several different places in a program, each with a different, but usually similar, meaning.

Local labels can be defined as often as they are needed, and will only affect the small area of the program they work with.

Not all Apple assemblers let you separate global and local variables, although "new" EDASM gives you a way to do this. Thus, each time you use some code module, you may have to pick a unique and different label name. Obvious ways to beat not having a local label capability are to number labels sequentially, such as START1, START2, START3, or to use creative misspelling, such as PRINT, PRIMT, PRENT.

Yet another way to classify assemblers is whether they generate *relocatable code* or not . . .

> **RELOCATABLE CODE ASSEMBLER—**
>
> **An assembler that generates special machine language code that will reposition itself anywhere in memory before use, reliably and automatically.**

Normally, your "typical" machine language program is only allowed to sit at one exact place in memory and has only one legal starting address. This is fine, if you always know where you want your program to go.

Dino machines often speak of *virtual memory*. One of the key features of virtual memory is that any program or any program module can go at any place in memory and still work. This gets real handy when you are tacking a bunch of mix-and-match machine language modules onto the top of an Applesloth program. Virtual memory is so powerful that you can easily think of a dozen more ferinstances where it sure would be nice to put anything anywhere and still have it work.

For micros and personal computers, relocatable code is a powerful idea whose time has come.

You can write machine language programs that can go anywhere in memory, so long as the code never calls itself or refers to itself with any absolute addresses. This means no absolute self-references such as loads or stores, no jumps, and no subroutines. This would be a simple example of a program that is self-relocating and can run anywhere. The disadvantage, of course, is that you aren't allowed to use most of the useful or interesting 6502 op codes when you try this.

Or, you can write a long and fancy machine language program that first finds out where it is sitting in memory, and then changes itself so it will run in its present location. The standard Apple II way of finding out where a program sits is to jump to a subroutine in the monitor with a known immediate RTS return, and then dig into the stack to find the calling address. From this point, you can play games that self-modify the rest of the code so it works where it is sitting.

The Apple people have gone one step better, and now have "R" files. These "R" files are *relocatable* code modules, that work with special loader software to put a machine language module anywhere in memory. They do this by dragging along a data table that lists everything that references absolute locations. These locations are changed as needed.

If you want to use "R" files to make your machine language code relocatable, then you have to use an assembler that can handle relocatable code.

Relocatable, or "R" files are nice for advanced programming concepts, but let's get back to some more simple mainstream stuff.

Another way to classify assemblers is by *where* they put the machine language program they generate. You have a choice of *in-place* or *disk-based* assembly . . .

IN-PLACE ASSEMBLER—

An assembler that assembles its machine language code directly into RAM memory.

DISK-BASED ASSEMBLER—

An assembler that assembles its machine language code onto a disk-based file.

An in-place assembler will directly assemble its machine language code into the RAM of your Apple II or IIe. This is fast and convenient. Often, you can test your machine language program immediately, without needing any reloading or rework. You could even get by without any disk drives, although no serious assembly language programmer would ever consider this.

There are several disadvantages to in-place assembly. You are limited to shorter programs, since both the assembler program and the final machine language code must fit into memory at the same time. The machine language program may have to be moved so it can run, if the intended place for the final machine language program conflicts with the code space needed for the *co-resident* assembler program.

A disk-based assembler reads a disk file as an input and generates a different disk file as an output. The files can be much longer than the space available in memory, since all the assembler has to do is keep a short stash of labels and cross-references handy. Thus, you can easily write and assemble very long programs with a disk-based assembler.

There are also no limits to where the final program code sits, since this is code stashed on a disk, and not code stuffed into the machine. You can easily assemble a program that must sit in the same space the assembler does; can overwrite text screens; can work on page zero, the stack, the keyboard buffer, or wherever. The final code is ready to use without any relocation.

The bad news here is that disk drives tend to be very slow, and that you have a long song and dance to go through when you want to test your machine language program, since you may have to get out of the assembler program, and then load and run your machine language program. The "test-modify-reassemble" round trip time can be much longer with a disk-based assembler.

Newer DOS speedup tricks ease the turnaround time involved in disk access. Fast loading and storing of text files by modified DOS code helps bunches.

Another factor that makes this long round trip time not too bad is that many programmers, including myself, are running a printer most of the time that they are assembling. This slows down an *in-place* assembler to where it is almost as infuriatingly slow as a disk-based assembler can be. If you happen to be using a daisywheel printer for quality output, there really isn't that much round trip time difference

between an in-place and a disk-based assembler. A print buffer or a spooler can speed things up a whole lot for either type of assembler.

Some in-place assemblers give you the option of assembling to disk, and vice versa. This can give you the best of both worlds.

Here's two more terms for you . . .

> **MODULAR ASSEMBLER—**
>
> **An assembler where editing and assembly routines are separate modules, only one of which is loaded into the machine and used at any particular time.**
>
> **CO-RESIDENT ASSEMBLER—**
>
> **An assembler where editing and assembly routines can both be present in the machine at one time.**

Both have advantages. Modular assembly gives you more room for your source code and possibly for in-place object code as well. The modular routines can also be longer and fancier, since they have more "elbow room" in which to work. Co-resident assembly is faster and shortens the edit-assemble-test round trip time considerably.

That covers most of the more important ways to classify assemblers. Sometimes you might get involved with a *cross assembler*. . .

> **CROSS ASSEMBLER—**
>
> **An assembler that is displeased or otherwise unhappy with the inane garbage it is being fed.**

Uh, whoops. Computer error. Let's run that one by again . . .

> **CROSS ASSEMBLER—**
>
> **An assembler that runs on one computer system, but generates machine language programs for a different computer system, possibly even using a different microprocessor.**

If you are only using an Apple II to assemble 6502 machine language programs that are only to run on an Apple II, then you will most likely never need a cross assembler.

Cross assemblers work on one machine but generate code for a different one. For instance, you could use an Apple II to generate machine language programs that are ready to run on simpler 6502 machines, such as the KIM, AIM, and SYM gang or for a 6502 controller card. This gives you all the full resources of your Apple, including disk drives, modem, printer, et al., to let you develop programs for these other machines. If you teach your Apple II to generate cassette tapes to the standards of these other machines, you can directly

download programs from the Apple to the target machine. Or else use serial ports to exchange programs and data.

Other cross assemblers may work with different microprocessor families. Thus, by going to the right kind of *emulator* software, you can use an Apple to generate TRS-80 code, 68000 code, MacIntosh routines, CRAY 1 code, or anything else you like.

Many Apple-based assemblers will provide modules to let you do cross assembly. The S-C Assembler modules in particular let you cross assemble into dozens of different microprocessor CPUs or even into dino minicomputers.

That pretty much rounds out our survey of the types of assemblers that you might find interesting or handy to use. Our main interest in the rest of this book, though, will be in using a disk-based full assembler to generate Apple II programs for Apple II or IIe use.

HOW ASSEMBLERS WORK

A miniassembler works as if it were an "automated pocket card." You pick a starting address, and start punching assembly language mnemonics into the machine. The miniassembler then converts these mnemonics into the correct op codes for you.

All the op codes are figured out automatically. Different address modes are entered by special symbols following the mnemonics. Relative branches are also automatically calculated, although you do have to take a guess at forward branches and then "repair" the guess when you get to the place in the program the branch is supposedly going to.

A miniassembler only works on one mnemonic at a time, and has no way of remembering what it did before or anticipating what it will do in the future. There are no labels, limited comments, and limited printed records. There is no record of what goes into the miniassembler, unless you create one yourself using some programming form, a disassembly listing, or possibly a word processor.

The miniassembler is an essential "go-between" step that should separate your first hand-coded machine language programs from your use of a full assembler. A miniassembler gives you insight into what the assembly process is all about, drives home the need to understand address modes, and forces you to become a better and more thoughtful machine language programmer.

Most serious programmers soon demand much more than a miniassembler can deliver. So they step up to either a full assembler or a macroassembler.

Both these work pretty much the same way. All a macroassembler does is give you some more features and some extra bells and whistles to make your programming efforts more legible and more convenient.

Think about how you hand code a machine language program. First you decide what you want to do. Then you actually do the encoding process to come up with the correct op codes and addresses.

Full assemblers work the same way. First you write a script that will tell the assembler exactly what it is that you want done. Then you feed the script to the assembler, and it takes the commands in that *script*, and then goes ahead and tries to build a machine language program, following your instructions.

There are two stages involved in using an assembler . . .

> **TO USE A FULL ASSEMBLER**
>
> FIRST, you write a script or a series of instructions telling the assembler exactly what it is that you want done.
>
> SECOND, you send this script or series of instructions to the assembler so the assembler can use these instructions to write your machine language program.

You go to the assembler and say "Here is what I want done." The assembler then takes this listing of what is to be done and then actually tries to do it, generating you a machine language program.

The script, or series of instructions is called the *source code* . . .

> **SOURCE CODE—**
>
> The series of instructions you send to an assembler so it can assemble a program for you.
>
> Source codes are written more or less as an English text, but there are stringent use rules that must be EXACTLY followed.

The assembler then reads your source code and tries to make some sense out of it. If you obey all the rules, the assembler will take the instructions in the source code, and follow its built-in rules so it can generate a machine language program for you. This generated program is called the *object code* . . .

> **OBJECT CODE—**
>
> The machine language program or other code that the assembler produces for you, following the instructions in the source code.
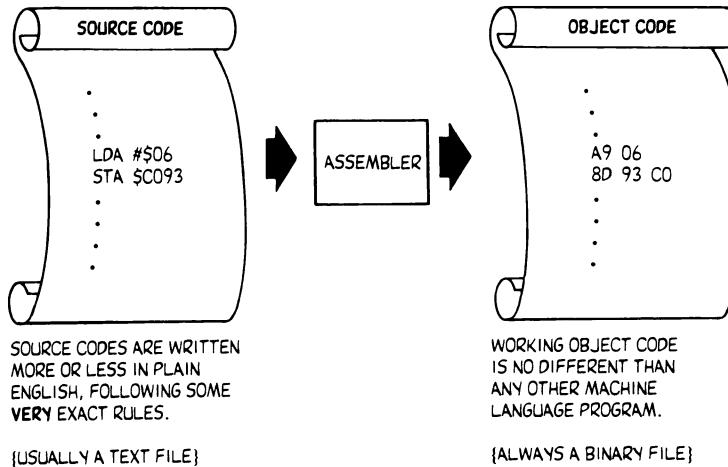
You write a script called the source code. The assembler then takes the source code and converts it into a machine language program called the object code.

Like so . . .

HOW TO TELL SOURCE CODE
FROM OBJECT CODE:

THE **SOURCE CODE** IS A
SERIES OF INSTRUCTIONS YOU
WRITE THAT TELLS THE
ASSEMBLER WHAT TO DO. . .

THE **OBJECT CODE** IS A
MACHINE LANGUAGE PROGRAM
OR CODE MODULE THAT THE
ASSEMBLER BUILDS FOR YOU
FOLLOWING THE SOURCE CODE
INSTRUCTIONS. . .

```
      SOURCE CODE
         .
         .
         .
      LDA  #$06
      STA  $C093
         .
         .
         .
```

⮕ ASSEMBLER ⮕

```
      OBJECT CODE
         .
         .
         .
      A9 06
      8D 93 C0
         .
         .
         .
```

SOURCE CODES ARE WRITTEN
MORE OR LESS IN PLAIN
ENGLISH, FOLLOWING SOME
VERY EXACT RULES.

WORKING OBJECT CODE
IS NO DIFFERENT THAN
ANY OTHER MACHINE
LANGUAGE PROGRAM.

{USUALLY A TEXT FILE}

{ALWAYS A BINARY FILE}

Source code is sometimes called the *source file,* and object code is sometimes called the *object file,* particularly on disk-based assemblers. Either way, the source is your script, and the object is the machine language result.

Should files or tables of data also be needed, a good assembler will also produce these for you, starting with ASCII values or a string of hex or decimal numbers.

Note that the source code and the object code are totally different types of beasties. The source code is a series of English-like instructions that you wrote. The object code is the machine language program the assembler has generated for you . . .

> **Source code files and object code files are totally different.**
>
> ## DON'T MIX THEM UP!
>
> **Source code = your instructions**
> **Object code = assembly result**

In the assembler we will be using in this book, the source code is usually stored on the diskette as a *text* file. The object code, of *course,* must be stored as a *binary file* since it is a runnable machine language program or some part of one.

Other assemblers may store their source code as a binary file, as a

text file, or may use some special format. But, no way will *any* source code run as a machine language program, ever!

So, obviously . . .

> **Source and object code files MUST ALWAYS have different names!**

The reason for this, of course, is that a source file is one thing (instructions from you), and an object file is something entirely different (machine language code the assembler generates). If you give these totally different code files the same name, then you'll get into the same troubles you would if you put two identical names on any pair of files on the same diskette.

There are at least three ways you can name source and object files so you can tell they belong together, yet still separately recognize them. One way is to add something to the source name to say it is indeed a source file. A second method is to add something to the object name to say it is obviously an object file. The third route involves prefixes.

For instance, if you are working with a program called SNARF, you might call your source code SNARF 1.0.SOURCE, and your object program SNARF 1.0. I like this route, since your final machine language program is properly named for final use.

One other alternative is to call the source program SNARF 1.0 and the object program SNARF 1.0.OBJ0. This is the "default" way the assembler we will use in this book does things, so apparently someone somewhere must like this strange notation. Other assemblers might drop the version count following ".OBJ".

Others prefer to use prefixes, such as S.SNARF for a source code file and BSNARF for the binary object file.

You should always keep track of the version of a program by adding numbers to the name . . .

> **Add version numbers to all your programs, always making the latest program have the highest number.**

The usual way you do this is to call your updated programs SNARF 0.1, SNARF 0.2, SNARF 0.3, and so on. Use tenths for small changes and routine updates. Use tens or hundreds for major overhauls. For instance, you keep adding SNARF 0.4, SNARF 0.5, and so on for small changes or improvements. Should you "refocus" your program into something wildly different, start over again with SNARF 10.1, SNARF 10.2, and so on.

NEVER overwrite the last working copy of anything you have . . .

> **NEVER overwrite the last good copy of your source code!**
>
> **ALWAYS add a new version number higher than the previous ones.**
>
> **Delete old code ONLY when it is many versions behind and cannot possibly have any more use.**

In short, back everything up six ways from Sunday. Don't throw anything old away till you desperately need disk room. Even then, be very careful and save a printed record.

Sometimes when you think you are "improving" source code, you may actually be destroying it, or else throwing the baby out and keeping the bath water. There's also the random glitch that destroys a file. Either way, with no backup, you end up in deep trouble.

Most full assemblers will have a program that will make writing the source code script easy and fun to do. This part of an assembly system is usually called the *editor*. The editor is pretty much a word processor program that is set up to exactly follow the rules needed by the part of the assembly system that is to generate the machine language result for you.

The actual part of the assembly system that takes the edited source code script and converts it into a machine language program is called the *assembler* . . .

> **EDITOR—**
>
> **That part of an assembly system which helps you create or modify a source code file.**
>
> **ASSEMBLER—**
>
> **That part of an assembly system which takes the source code instructions and converts them into machine language object code.**

This gets sticky fast, for "editor" can mean two different things and "assembler" can mean two different things.

When you are talking about the big program, most people say "assembler" when they really mean "assembly language development system." Two important parts of most typical Apple assembly language development systems are a way to create and modify source programs, called the editor; and another separate way, called an assembler, to take the source code file and generate an object code file.

To further foul up the works, the editor part of the assembly language development system is used both to create and modify source code files. The process of using an editor to create a source code file is sometimes called *entry*, while the process of using an editor to change an existing source code file is sometimes called *editing*.

You can also do your editing and entry with a word processor, in which case you can call what you are doing anything you care to.

The context usually will help you out. An assembler is either the whole development system, or else just that part of the development system that does the actual assembly. An editor is either the whole module that lets you create or modify a source code file, or else just that part of the program that is actively involved in changing or correcting an existing file.

Arrgh! Anyway . . .

The way a full Apple assembler works is that you first use the editor part of the development system to create a source code file. As a reminder, this source code file is a series of English-like instructions. You might also use the editor to change or correct existing source code.

Then it is the assembler's turn.

The assembler reads the source code file several times. Each time is called a *pass,* and most assemblers take at least two passes to complete the assembly process. The assembler goes all the way through the source file to find all the labels, all the definitions, and any forward branch references. It saves this data in suitable tables or lists. Then, the assembler makes a second pass to convert all these references into useful machine language object code.

You first write source code, and then have the assembler assemble it for you. Then you test the code. Should you not like the results, you go back and change the source code to correct any mistakes you made or make other improvements.

The edit-assemble-test process goes round and round many times. It is not unusual to need dozens or even hundreds of cycles through the works to get what you finally want.

Most assembler programs will generate error messages for you. An error message simply means that what you sent the assembler was so stupid that it couldn't figure it out. Naturally, you can feed the assembler perfectly correct instructions that will still generate worthless or nonworking code . . .

The only thing a message of—

"SUCCESSFUL ASSEMBLY: NO ERRORS"

tells you is that you haven't done something so incredibly stupid that the assembler couldn't figure out what it was you were trying to tell it to do.

It is very easy to successfully assemble totally worthless code.

You will definitely be seeing much more on error messages. Will you ever. The useful thing about error messages is that they will point directly to the place in the source code where any problems were found that are so bad the assembler cannot, or should not, continue.

To recap, full assemblers and macroassemblers consist of at least two related programs. One program, called an editor, lets you create a script or source code file that explains what it is that you want done. A second program, called the assembler, then makes the several

passes through the source code file, and converts these instructions into a machine language code object file.

If the assembler finds any really bad problems, it will give you error messages. Some of these errors will stop the assembler dead in its tracks; others are just brought to your attention for a later repair. But, a lack of error messages in no way means that your final machine language program will work or that it will do what you want it to.

As we'll find out later, you can also use a word processor as a "new way" to create and edit source code, compared to the "old way" of using the editor in the assembler package.

By the way, on non-Apple or non-6502 machines, an "assembler" may be just that—a way to assemble programs with no provision whatsoever for entry or editing of source code. Beware of this dino trap.

WHICH ASSEMBLER?

Very simply, there is no single "best" assembler for the Apple II or IIe, nor is there likely to ever be one. You find a package that suits your needs and your personal programming style, and then go with it.

Here are several of the more popular Apple II and IIe assembly development systems . . .

SOME APPLE II AND IIe ASSEMBLERS	

ALDS
  author unknown
  (Microsoft)
	APPLE ASSEMBLY SYSTEM
	  by Paul Lutus
	  (Hayden)

APPLE EDASM ASSEMBLER
  by John Arkley
  (Apple Computer)
	ASSEMBLY SYSTEM
	  author unknown
	  (Stellation Two)

BIG MAC
  by Glen Bredon
  (A.P.P.L.E.)
	BOOTHWARE 8073
	  author unknown
	  (Microbasics)

EDIT 6502
  by Ken Leonhardi
  (LJK Enterprises)
	MERLIN
	  by Glen Bredon
	  (Southwestern Data Systems)

LISA
  by Randy Hyde
  (Lazer Systems)
	ORCAM
	  by Mike Westerfield
	  (Hayden)

S-C MACRO ASSEMBLER
  by Bob Sander-Cedarlof
  (S-C Software)
	THE ASSEMBLER
	  by Alan Floeter
	  (Micro-Sparc)

THE CHEAP ASSEMBLER
  by John Cox
  (Thunder Software)
	WELL TEMPERED ASSEMBLER
	  author unknown
	  (Avocet Systems)

We will put all the addresses and phone numbers in Appendix B to keep things orderly.

The Apple EDASM assember is really three different assembly packages. "Old" EDASM was written by Randy Wiggington and has been around for quite a while. There are two "new" EDASMs, both written by John Arkley. One "new" EDASM is DOS 3.3e based. The second one is ProDOS based. All three EDASMs are "alike but different somehow." See Appendix A for a summary of the key differences. Both "new" EDASM versions are available as toolkits in Apple's *Workbench* series.

Of the others listed, *Boothware 8073,* Avocet's *Well Tempered Assembler,* and Stellation's *Assembly System* are specialized cross assemblers, while most of the others are general-use full or macroassemblers. *Merlin* is an enhanced commercial version of *Big Mac.*

The price of these assembly systems presently ranges from $22 to $400. Very interestingly, the *value* of each of these assembly systems is almost a perfect *inverse* of their pricing! Thus, the *more* you pay, the *less* you get. I guess it was bound to happen sooner or later.

For someone else's opinion of these programs, check into *Peelings* Volume 3, Number 2, February, 1982. More current reviews are also likely to appear in *Peelings* and *Infoworld* as well as in all the usual Apple magazines and review anthologies.

We are purposely not going to give you a blow-by-blow comparison of all these different assemblers. Instead, we are going to use Apple Computer's own recently upgraded and overhauled assembler for the rest of the book. This one is called *The Apple 6502 Editor/Assembler,* or *EDASM* for short, and is found on one of two popular utility diskettes in the *Workbench* series. Both DOS 3.3e and ProDOS versions are available. These diskettes cost around $75, but since there are lots of other goodies on the disks, particularly BUGBYTER and HIRES character generator systems and new character fonts, your actual cost for the assembler will be much less. Unbundled, EDASM is the cheapest assembler available.

Why this assembler? Well, first, I like it. Secondly, I use it for all my work, and it is the one I know best and have used the most. We also use it for commercial program development here at *Synergetics,* and for several microprocessor courses over at EAC, our local community college.

EDASM is probably the most popular assembler, if for no other reason than there are great heaping bunches of copies of the *DOS Toolkit* in circulation. EDASM is normally a disk-based assembler, so it can handle programs of any length, particularly very long ones that cannot easily be handled in one piece by the others. EDASM also does relocatable code assembly very well.

EDASM's recent overhaul now includes macros, in-place assembly, optional ProDOS compatibility, co-resident assembly, along with many other new and most useful features. Important differences between "new" and "old" EDASM are summarized in Appendix A.

And, programs written under EDASM, seem to me to be much "cleaner," much easier to read, and much more well thought out and better documented than some of the others I have seen that use competitive assemblers. This, admittedly, is very subjective. It may just be

that more people are using EDASM, or that I am looking in the wrong places.

Naturally, the "best" software is almost always available from sources *other* than Apple Computer. This goes without saying. But I have yet to find anything unacceptably bad about new EDASM. Incidentally, others consider the *S-C Assembler* the "best" available, no holds barred, while *Big Mac* is often rated as the "best bargain."

Critics are quick to point out that EDASM has some limits to its macros and cannot easily separate global and local variables. They delight in EDASM's much slower speed and painful reloading when it is not in its in-place assembly mode.

There are also some minor peeves, such as needing an extra "A" at the end of accumulator mode addressing, inconsistency between how you exit the entry and editing modes, some overly strict page zero addressing rules, and a printer bug that sometimes messes up the top line of continued listings.

You can minimize the impact of these disadvantages. For instance, you can fake almost all of the things an in-code macro is supposed to do by building up a source code macro library on your diskette, and pulling off these modules as needed. Most of the time, many assembly language programmers will keep their printer running during an assembly. This way, you always have a printed record of exactly what you have at any time. If you do keep your printer on, the disk-based assemblers really aren't that much slower than any other, since the printer is usually holding up the works. A spooler or a print buffer could, of course, be added to speed things up.

And, yes, EDASM's editor is dismal, dreary, and dumpy. Putrid even. But, as we'll find out later, you simply do most of your entering and editing with *Apple Writer IIe* instead, and handle some of your macros with a glossary or else with WPL. Which instantly converts one of the worst editors into one of the most powerful available. More on this in chapter five.

Anyway, I like EDASM, and I use it a lot, and we are going to use it here.

But . . .

> **Do not EVER buy ANY assembler program until you have had a long talk with someone who believes in and consistently uses that program!**

The main impact of new EDASM on this book will take place in the chapters that follow. Since any assembler has to do the very same things that EDASM does, you should be able to edit these chapters with margin notes any time you find differences between the details of how EDASM works and how your assembler works. We'll even give you extra room for this every now and then. All the detailed ripoff modules should work with any full or macroassembler of your choosing.

TOOLS AND RESOURCES

One assembler program and one assembler book will in no way make you a decent assembly language programmer.

I have yet to see a decent Apple II assembly language programmer who was proud of the work he did last *week*, let alone last year. Assembly language programming is a continuous learning and skill-building process.

So, those who think they are going to instantly become fantastic assembly language programmers are both deluding themselves, and ripping off their customers as well. If you are unfortunate enough to ever meet one of these dudes, please go out of your way to talk him into writing programs for non-Apple machines. Send him to *Honeywell*. Teach him COBOL. You will kill two birds with one stone.

Instead . . .

> **The only way you can become a halfway decent machine language programmer is through lots and lots of practice and much hands-on experience.**
>
> **The time frame involves years, and not just days, weeks, or months.**

But, as someone once said, "The longest journey starts with a single step." If you want a shot at the brass ring and want to join the club, you gotta start somewhere, sometime. Like now. That's a mighty big bag of nickels up for grabs.

Maybe some time can be saved by showing you what assembler system I use and what tools and resources I work with. One way to find out.

Here's two possible assembly language programming setups . . .

> **ASSEMBLY LANGUAGE WORK STATIONS**
>
> Apple II Computer with 48K RAM
> Integer ROM set in mainframe
> Absolute reset ROM in mainframe
> Applesoft ROM card with Autostart
>
> -or, preferably-
>
> Apple IIe computer with 128K RAM
> and custom "absolute reset" EPROM
> monitor ROM
>
> -plus-
>
> Two disk drives
> Quality daisywheel printer
> Metal printwheels

Some comments on these arrangements. First, it is absolutely essen-

tial on older Apples that you have an "old" monitor ROM in your mainframe, if you are at all serious about assembly work. Besides the very handy single-step, trace and debug features, this old ROM lets you stop *any* program at any time for any reason, under absolute control. The Integer ROM set gives you the old miniassembler, the programmer's aid, and access to the "Sweet 16" pseudo 16-bit machine routines, along with the old floating point package.

The Apple IIe is, of course, a much better choice for developing newer software. But you will definitely want to provide your own custom monitor EPROMs to pick up absolute reset and eliminate the obscene "hole-blasting" restart of the stock monitor chips. While you are at it, throw in a 65C02 as a new CPU, since these do so much more so much better.

To repeat . . .

If you are at all serious about assembly language programming, you MUST have a way to do an absolute and unconditional reset.

On older Apples, this takes the "old" F8 monitor of the Integer ROM set in the mainframe.

On the Apple IIe, this takes a pair of custom 64K EPROMs that replace the existing monitor.

On older Apples, you can either use a ROM card to pick up the Applesoft ROMs and the autostart ROM, or else go to a RAM card and Applesoft software. The RAM card is probably the better choice today, but should be modified for absolute hardware control. By the way, old monitor ROMs are often advertised in *Computer Shopper,* usually for $10 or less each.

A second disk drive is handy and almost essential. These days, you can get good drives much cheaper from sources other than Apple. I use a *u-SCI* as my second drive. Sometimes you can hold off on tasks that really need two drives and borrow a second drive just long enough to get the job done.

One useful advantage in mixing your brands of disk drives is that they will *sound* different while running. If you ever activate the wrong drive, this "aural feedback" makes it known pronto.

A dot-matrix printer is probably the best choice for writing and debugging programs, because these printers are very fast. But, it is absolutely inexcusable to *ever* publish *any* dot-matrix listing, even if your uncle is an optician . . .

Don't EVER publish ANYTHING that you have printed on a dot-matrix printer!

Now, there are a few people around who claim they can *actually* read a dot-matrix printout, particularly from newer model printers. This peculiar genetic deficiency usually shows up in inbred generations of dot-matrix printer salesmen, but is thankfully rare otherwise.

Unfortunately, the printing processes in use today cannot read or accept dot-matrix printout. By the time your dot-matrix listing goes through a bad ribbon, gets reduced, is photocopied, gets burned onto a plate, and finally gets printed, you will end up with a royal mess.

So, I use an older *Diablo* 630 daisywheel printer myself, since I can't justify having one printer for listing and debugging, and a second to generate camera-ready copy.

Of course, you use a film ribbon, and for your final copies, you use single-strike film. Naturally, it is totally inexcusable to ever retype or typeset an assembly listing, because errors are *certain* to be added. Errors that are hard to find and harder to correct.

By the way, every now and then some turkey will try to tell you that you cannot tell the difference in print quality between a metal and plastic printwheel. This is true only if (A) you are at least a thousand feet or more away from the page, and (B) you are blind.

There is as much difference between a heavier metal printwheel and a plastic printwheel as there is between the plastic one and dot-matrix print quality. This is especially true if you use one of the "heavier" metal fonts not available in plastic, and do so on a freshly adjusted machine. I am kind of partial to the Titan 10 metal wheel myself for listings, and to the Bold PS wheel for everything else.

While we are on the lookout for turkeys, watch out for misleading speed claims on newer daisywheels. The newly discovered "words per minute" rating is *ten times* the industry standard "characters per second" speed rating. Thus, a daisy rated at "120" is much *slower* than one rated at "40."

Even worse, the term "letter quality" has been bastardized into "near letter quality" or "correspondence quality." Well, "letter quality" means "looks like an old mangy Selectric." "Correspondence quality" means "not quite totally illegible." And, the "near" in "near letter quality" means the same thing as "nearly" getting a job, "nearly" winning a contest, or for that matter "nearly" getting run over by a garbage truck. A suitable synonym for "near" is "ain't."

Summing up, if you can, use a *fast* printer for assembly development and checkout, but be sure to use a *good* printer for your final published listings.

So much for the system. One of the really great things about the Apple II is that it forms its own superb development system. Would you believe that other computer systems make you buy program development hardware that can cost tens of thousands of dollars?

Fortunately, all you need to write good Apple II programs is a good Apple II or IIe computer.

Working Tools

What about other tools? What else do you need? I'd call a tool anything you use or refer to while you are using an assembler to write machine language programs. Most of the tools that are useful are books of one kind or another. But the crucial difference between any *old book* and a tool book is that the tool book gets used over and over again, while any old book just sits on the shelf.

Anyway, here is a list of the tools I find handy . . .

**TOOLS FOR ASSEMBLY
PROGRAMMING**

6502 Pocket Card (Rockwell)
6502 Plastic Card (Micro Logic)
6502 *Programming Manual* (Rockwell)
6502 *Hardware Manual* (Rockwell)

Apple II Reference Manual (Apple)
Apple DOS 3.3. Manual (Apple)
Apple Assembler Manual (Apple)
Applesoft and Integer Manuals

Old Apple Red Book (Out of print)
Apple Tech Notes (IAC)
Apple Monitor Peeled (Dougherty)
What's Where in the Apple (Micro Ink)

Beneath Apple DOS (Quality Software)
Hexadecimal Chronicles (Sams)
Lancaster's Micro Cookbooks (Sams)
Enhancing Your Apple II (Sams)

Printer manuals and repair tools
Paper, ribbons, diskettes, etc.
Page highlighters, all colors
HIRES and LORES screen forms

A quiet workspace

Many of these tools are obvious. Once again, addresses and phone numbers appear in Appendix B. We won't show prices or version numbers, because both are bound to change. As a disclaimer, this list is my choice and what I use. There's lots more and lots newer stuff available.

Going down the list, a pocket card is far and away the single most important tool. Pocket cards give you quick answers to questions like "Can I load X, absolute indexed by Y?" (yes); or "Can I do an indirect subroutine call?" (no—but you can JSR to a JMP indirect); "Can I set the V flag?" (not with software).

The pocket card also tells you how long the instructions take to execute. This is essential knowledge for any program that involves critical timing, and can be handy in any program.

The 6502 plastic card is equally useful. I use both. You can write on the plastic card with a grease pencil, but you can't fold it and carry it with you.

The 6502 *Software Manual* is also indispensable. You simply cannot do any assembler work without this book. The book was first written by MOS Technology, Inc., and for its time was one of the *finest* technical manuals *ever* produced by *any* semiconductor house. It is a classic in every sense of the word. The old MOS Technology, Inc. copies were big and easy to read with rugged blue covers. Today's knockoffs

by Rockwell International and Synertek are smaller, have lower print quality, and are harder to read.

The *6502 Hardware Manual* in the same series isn't nearly as well written or understandable, but it is also an important tool for the assembler programmer.

You will want all the usual Apple manuals, particularly the IIe technical reference manual and the DOS manuals. The Applesoft and Integer manuals and tutorials will be handy if you are tying machine language modules into BASIC, rather than writing decent all-machine programs.

You will also want to get access to a copy of the *Apple Tech Notes*. This thick series answers many Apple-use questions and spells out all known Apple bugs to date. All *International Apple Core* (IAC) member clubs have a copy. Or, if you can find a reasonable Apple dealer, they might let you look at their working copy. You can also buy these tech notes, but they are expensive.

What you won't be able to buy is a copy of the *Apple Red Book*. This was the original Apple II system manual. Among its priceless goodies is a schematic that is orderly and function-oriented rather than the intentionally confusing mess shown in the pre-IIe manuals. You'll also find complete details on the *Sweet 16* sixteen-bit software commands, detailed miniassembler listings, the original tone subs, low-cost serial interfaces, and listings on the original floating point package. Very handy and very essential if you are still working with an Apple II or II+. You'll have to copy this one on your own, as the *Red Book* is definitely out of print.

Note that the Apple IIe technical reference manual does not come with a IIe and has to be ordered separately at extra cost. This manual is absolutely essential for IIe assembly language programming.

The *Apple Monitor Peeled* is a very dated book. But, I still find it useful to understand and use the monitor features, while the "must-have" *Beneath Apple DOS* gives a thorough treatment to the disk operating system. *What's Where in the Apple II* is a detailed address-by-address listing of all known major uses of all memory locations in the entire machine. There are two parts to the listing. One part is arranged numerically and the other alphabetically. These listings are an update and extension of the original that appeared in the August 1979 issue of *Micro*. I use both the original article and the new book, because the original is easier to use and only takes a few notebook pages.

The *Hexadecimal Chronicles* (Sams 21802) is a reference that instantly gets you from decimal to hex to Integer BASIC's inverted decimal, and back again, along with ASCII conversions, and includes a hex arithmetic and circular branch calculator, and bunches of other goodies. This one is most useful when you are tying machine language subs to BASIC programs, or are moving BASIC pointers around to protect or capture a machine listing.

Volumes I and II of my micro cookbooks (Sams 21828 and 21829) should be a good way to pick up the fundamentals of hand-coded machine language programming. This is done through a series of *discovery modules* that lead you through most op codes of the 6502. You must use these discovery modules or something similar before you can even think about working with assemblers.

The *Enhancing Your Apple II* series (Sams 21822, etc . . .) gives

you many examples of machine language program modules and use ideas. In particular, the "tearing" method in Enhancement 3 of Volume I is essential for any assembly language programmer, since it shows you an astonishingly fast and easy way to tear apart and understand unknown machine language code.

You will also want a *complete* set of maintenance manuals and repair tools for your printer. These are usually *not* provided with your printer purchase. Note that most printer people charge bunches extra for their real service manuals. Often these will be broken up into a spec manual, a repair manual, a spare parts list, a price list, and special tools.

There are many other programming aids, support books, utility diskettes, and so on that are heavily advertised. I find myself buying but never getting around to using these. Around 90 percent of what's available is less than useless, so always check with someone that believes in a product before you consider buying it.

Naturally, you'll need some diskettes, tractor paper, film ribbons, and all the usual stuff like this. A complete set of page highlighters are also essential to have on hand. These are very useful for identifying changes and corrections on printouts and are absolutely essential for the "tearing" disassembly method to work.

What About Machine Programming Books?

You'll find dozens and dozens of books around that claim to teach you all you will ever want to know about 6502 machine language or 6502 assembly language programming, and then some.

Usually, you buy these books by the running yard, with a price of $28 per inch or so being typical. Mill ends are slightly cheaper. Either way, put them on your bookshelf to amaze and astound your friends. Or, if you happen to have a table with a missing leg, put a stack of them to their only known practical use.

Very few people ever read these books. In fact, most of these books have been designed from the ground up so you could not possibly read and understand them even if you wanted to.

A very select few of these books are genuinely outstanding. Unfortunately, most of the others are utterly atrocious ripoffs. The trash-to-good ratio here is well over 30:1 and is steadily and appallingly climbing.

And even if everybody else thinks some title is a great book, it may not suit you, since its level may be too advanced, or too simple, or locked into some obscure trainer, or too pro-dino, or too far off in left field, or too whatever.

So, let us repeat what we said earlier about assembler programs, only this time we'll apply it to programming and assembly books . . .

> **The overwhelming majority of all programming and assembly books will NOT meet your personal needs.**
>
> **Do not EVER buy ANY assembler book or any machine language book until you have had a long talk with someone who believes in and consistently uses that book!**

The safest thing to do is to wade into the lair of your nearest Apple machine language freak and find out which books are out front, have torn or missing covers, and are thoroughly thumbed over. Don't even consider a book that has nothing spilled on its pages.

Come to think of it, though, it is *never* safe to wade into the lair of your nearest Apple machine language freak. Forewarned is forsooth, though, or whatever.

Far off.

At any rate, avoid buying these books unless you happen to want a complete set of "one of each." But that gets expensive in a hurry.

Software

The software you use will, of course, depend on which assembler you chose, and what else you decide to have on hand.

Here's what I usually work with . . .

SOFTWARE FOR ASSEMBLY PROGRAMMING
EDASM and BUGBYTER *System Master* *DOS Toolkit* *Inspector* *Bag of Tricks* *Apple Writer IIe* *Copy II Plus* *Enhancing Diskettes*

EDASM is the Apple assembly development system. Be sure to use one of the greatly improved "new" versions, either for DOS 3.3e or ProDOS. You should be familiar with the *System Master* diskette by now, particularly the program *FID*. The *DOS Toolkit* holds EDASM and BUGBYTER, along with several other interesting utilities and HIRES character fonts. *Inspector* is one of many available file utilities, while *Copy II Plus* is a versatile and informative copy and disk speed program. *Locksmith* is comparable.

Bag of Tricks is most useful for fixing bad diskettes. *Apple Writer IIe*, of course, is a great word processor and gets used for all your documentation, besides being a better source code editor than the one in EDASM. The *Enhancing* diskettes are from the *Enhancing Your Apple II* series.

There are a lot of new utility programs available today that do things like single-step whole programs, manipulate and search files, dump ASCII strings, disassemble listings, generate cross references, provide HIRES utilities, offer ampersand links, edit diskettes, and so on.

I haven't gotten around to trying very many of these. The obvious advantages of these new utilities are that they make writing and testing programs much quicker and easier. Two obvious disadvantages are that costs pile up at $30 and $100 per diskette, and that some of these programs may conflict with where you want to be in memory. A few of these are excellent; many others are less than useless.

Anyway, so much for the software. Use what you need and like.

Assembly Magazines

Magazines are some of the best places to learn about machine language and assembly language programming. Here's my choice of the best, arranged in order of assembly language usefulness or importance to me . . .

```
ASSEMBLY LANGUAGE MAGAZINES

Call A.P.P.L.E.
Apple Assembly Lines
Apple Orchard
Peelings

Hardcore
InCider
Cider Press
Nibble
```

The finest Apple assembly language magazine, no holds barred, is *Call A.P.P.L.E.* A complete set of these, their user library diskettes, and their publications is absolutely essential to serious machine language or assembly language programming. Their thick *All about DOS, All about Applesoft,* and *All about Applewriter* manuals are particularly valuable.

Apple Assembly Line is a funky little newsletter with fantastic vibes. It centers on the S-C family of fine assemblers, but is otherwise most readable. *Apple Orchard* is the IAC publication, and often has interesting reprints from the various newsletters. *Peelings* is the only source of well thoughtout and largely unbiased Apple software reviews. *Hardcore* has some very interesting stuff in it, but it could be so much more than it is. You'll also find very old issues of *Micro* to be most handy and informative, but this one clearly has peaked, so it is not on the list. *Cider Press* is a newsletter of the San Francisco Apple Core.

Note that this listing of magazines specifically involves assembly language programming. There are other great micro magazines, such as *Byte, Infoworld, Computer Shopper, Microcomputing, Creative Computing, Dr. Dobbs Journal, Softalk,* and dozens more. These have all sorts of useful things in them, but they do not consistently center on Apple II or IIe machine language programs and assembly language programming techniques. There are also hundreds of club newsletters out there, many of which will have new and useful assembly tidbits.

Unfortunately, the price of the club newsletters is high, and their quality is steadily dropping. The reason for this drop in quality is that most Apple clubs are now of, by, and for users, rather than hackers. This trend is intrinsically evil and despicable.

Also sad.

There's also a musical chairs game going on where everybody reprints everybody else without anything new ever being generated. Which waters down the stock something awful.

Since the trend is away from hackers, the older issues of most club newsletters will most often have the better goodies in them, so it pays to dig back. Way back. The *Denver Apple Pi* group maintains an

extensive on-line data base of virtually everything ever written in any Apple publication.

Reprints and Anthologies

Two other essential resources are reprints and anthologies . . .

IMPORTANT ASSEMBLER REPRINTS
Abacus Plus (ABACUS) *Best of Cider Press* (SFAC) *Inside Washington Apple Pi* (WAP) *Peeking at Call Apple* (A.P.P.L.E.) *Wozpack* (A.P.P.L.E.)

These reprints usually show off "the best of" some year's newsletter or magazine output. They are a good way to get everything at once fairly cheaply. The argument that you are buying old information is offset by the fact that older Apple information is often better and more recent information these days. The more blatant errors are likely to be corrected before reprinting as well.

We must also mention the *Apple Avocation Alliance* as well. These people stock just about every available public domain Apple program, and will copy them for you at a cost around twelve cents each. AAA also has terrific diskette prices. Unfortunately, except for several absolute gems, very few public domain programs are worth twelve cents apiece. Nonetheless, studying these may prevent you from reinventing the wheel and should clearly and concisely show you how *not* to do things.

Other public domain program libraries include the extensive ones offered by the *IAC, Call A.P.P.L.E.,* and the *San Francisco Apple Core.*

That's sure a long list of resources for assembly language programming. But, if you think that's bad, you should see all the garbage I bought and did *not* tell you about. Hopefully, these resource listings will cut down the totally ridiculous costs of getting into assembler work into costs that are just stupendous.

Naturally, don't buy anything you haven't looked at first. Work with club groups. Check into schools. Ask friends. Visit company and technical libraries. Pick up whatever works for you.

But don't try to write Apple machined language programs in a vacuum. That may have worked in 1977, but no more. Those days are long gone.

Get and stay informed.

DISASSEMBLERS

You mean that after you go to all the trouble to assemble a program, that you may want to take it all apart again?

You better believe it.

The opposite of assembly is called *disassembly.* You disassemble a program or listing when you want to find out what the code is trying to do or how it is supposed to work . . .

DISASSEMBLER—

Any tool that lets you take apart a machine language program to see what is in it or how it is supposed to work.

Naturally, it is totally inexcusable to *ever* buy or use *any* piece of Apple software without completely tearing apart the program to see what is inside and how it works. Also, naturally, the first thing you do to any locked program is make yourself several unlocked copies under standard DOS. Then generate your own modifiable assembler source code, and a complete set of working source files.

You should do this automatically the first time you boot any new disk. Every time.

Without fail.

Far and away the best way to learn assembler programming techniques is to diligently study how others do it . . .

Far and away the best way to pick up assembly language skills and new use ideas is to . . .

TEAR APART EXISTING PROGRAMS

There are several good ways to disassemble existing code. Here are the three I normally use, in order of increasing complexity . . .

WAYS TO DISASSEMBLE CODE
1. Use the disassembler in the Apple system monitor or BUGBYTER.
2. Use the "tearing" method from the *Enhance* series.
3. Use a capturing disassembler, such as Rak-Ware's DISASM.

There is a "L" or *List* command in the Apple system monitor that will disassemble any program for you twenty lines at a time. For more lines, you use more L's. This disassembling lister converts object code into assembly mnemonics and shows such things as the address modes and the absolute addresses that relative branches go to. There are no labels or comments. For a printed record, you simply turn the printer on before you list the lines you want disassembled.

The "tearing" method appears in Enhancement 3 of Volume I of the *Enhancing Your Apple II Series* (Sams 21822). This gives you an astonishingly fast and easy way to tear apart any unknown machine language listing and provides for full comments and accurate labels.

A capturing disassembler tears apart object code and then converts it into a source code file that EDASM or another assembler can use. It puts labels on everything needed, but these labels are simply coded

sequentially. You then have to go through the listing and add your own comments and make all the labels more meaningful. Sometimes, you can predefine useful label names. A capturing disassembler usually includes a complete *cross reference* of who refers to whom when.

The DISASM program by Rak-Ware is the only one of these I have worked with so far. Similar products are available from Decision Systems and Anthro Digital.

DISASM does what they say it will and is reasonably priced. Their cross reference generator is particularly useful. An alternative to a disassembler is to rekey the entire results of the "tearing" method. Which is the better route depends on your programming style and the length of the program under attack. Using "new way" editing does simplify and speed up the repairs to a captured listing.

DISASM's triple cross references are most useful, though. You get internal, external, and page zero reference tables that are absolutely essential to tearing apart any major listing. Very nice.

Regardless of which disassembly method you use, there is one big gotcha you must watch for . . .

> **A disassembler will only give you useful results if it is working on VALID code, and then only when begun at a LEGAL starting point.**
>
> **Otherwise you get garbage.**

What this says is that you can only disassemble code that has been previously assembled. Try to disassemble a file or some data values, and you get bunches of question marks and totally absurd op codes.

Even with legal and working code, you also have to start at the right place. If you have a three-byte instruction, you must start on the first byte. Start on the second or third byte and you get wildly wrong results. Having the wrong starting point in legal code isn't nearly as bad as trying to disassemble a data file or a text file, since the disassembler will probably straighten itself up and fly right after a few wrong listings. But watch this detail very carefully.

If you try to disassemble, say, an ASCII file instead of legal op codes, your cross references will end up giving you bunches of illegal and nonexistent "artifacts," caused by reading pairs of ASCII characters as addresses.

For instance, an "AB" ASCII pair may generate a false address of $4241, and so on. You will also get cross reference artifacts generated if there are short stashes or other files buried inside your legal op codes. These artifacts can be eliminated one at a time by hand, or by telling the disassembler to "skip over" one or more tables.

You need both assemblers and disassemblers to do a decent programming job. One puts together, the other takes apart.

WHAT AN ASSEMBLER WON'T DO

A car is one possible way to go to the bakery, get a loaf of bread, and then return. But there is no way a car can do this *by itself.*

You have to drive the car.

In the same way, an assembler is a great tool to *help* you write machine language programs, making the process easy, fun, powerful, fast, and convenient. But there is *no way* that an assembler will write programs for you.

You have to tell an assembler *exactly* what it is you want done, *exactly* when you want it done, and *exactly* how to do it . . .

An assembler will NOT write machine language programs for you!

You must tell the assembler ahead of time exactly what it is you want done, when you want it done, and how it is to be done.

Thus, an assembler is nothing more than a very powerful tool that will do exactly what you tell it to. To use an assembler, you *must* already be a competent and knowledgeable machined language programmer.

To get into this game and go for the brass ring, you *must* start by hand coding and hand debugging a few hundred lines of machine language code on your own. Then you should get with a miniassembler and practice with it, again for several hundred more lines.

Next, you should use the "tearing" method to take apart and study at least a dozen major winning Apple programs. This shows you how the "big boys" do it. Finally, if and only if you thoroughly understand what machine language is all about, you should move up to a full assembler or a macroassembler.

Any other way isn't even wrong.

2

SOURCE CODE DETAILS

If you are going to have an assembler or an assembly language development system create a machine language program for you, somehow you have to give the assembler some instructions.

Once again, there is no way an assembler will write a program for you. All an assembler can do is take the exact instructions you give it and then begin from there to try and come up with some useful code.

We have seen that these exact instructions are called the *source code* . . .

> **SOURCE CODE—**
>
> **The series of instructions you send to an assembler so it can assemble a program for you.**

You can think of the source code, or source code file, as a script or a series of instructions. In this script, you will usually find op codes and "how?" or "with what?" qualifiers that go with the op codes as needed for certain address modes.

Instructions to create subroutines and data files may also be included. You most likely will also find special instructions that vaguely resemble op codes that are intended for use by the assembler, rather than becoming part of the final machine language program. We'll find out later that these are called *pseudo-ops*.

In the script, you will also find definitions of labels and values. There will also be lots of *comments,* or user documentation. Comments can include such things as a title block, the copyright notice and author credit, a description of what the program does, instructions on how to run the program, and listings of any gotchas or any modifications that might be needed. Parts of the script will also be involved in the *pretty printing* that makes the entire script easy to read and easy to use.

Examples of pretty printing are blank lines, page breaks, and spaces used for clarity or centering.

To sum up, a script or source code file contains all the information needed for the assembler to put together a useful machine language program for you, along with all the documentation needed to tell people what is happening in the process.

In this chapter, we will find out just what source code is and how to use the "work unit" of the source code file, which is called a *line.* After we pick up these internal details, we'll go on in chapter three to find one possible way to organize and structure your source code.

Then, with this background, we'll go on to chapters four and five. Here, we'll see how you actually go about writing and then editing, or changing, a source code file for the assembly language program of your choice. Chapter four will show us the "old way" of using an editor in its intended way, while chapter five will give us full details of the "new way" of using a word processor instead.

The foremost use rule involving source code is . . .

The source code file is more or less a series of instructions in plain, old English, except . . .

ALL USE RULES MUST
BE EXACTLY FOLLOWED!

There are some very exacting and very nit-picking rules as to what goes into the script. Disobey these rules, and the assembler will generate garbage for you or simply will not work at all.

In particular . . .

Simple things like a missing or an extra space or a forgotten "$" for hex symbol can make the entire source code totally worthless!

Source codes are most useful, handy, and informative. But, you absolutely MUST follow the exact use rules involved with source code files if you are ever going to get anything usable out the other end of the pipe.

SOURCE CODE FILE FORMATS

The source code file will hold enough information to do the job you want done. The length of your source code can be just a few characters for a simple patch, through part of a page for a minor subroutine,

or many dozens of pages for an elaborate or very long, full-blown machine language program. The source code will do what you want it to. You make it as long as you need to handle the task at hand.

Some assemblers put a limit on how long the source code can be. If this happens, you break the source code into logical chunks and process one chunk at a time. Then you take the machine language modules you get from this process, and recombine them into a single, long program.

The EDASM assembler we will use as our "baseline" assembler is usually disk based, and lets you write very long programs in one piece if you want to. Often, though, it is best to work in small and separate modules of your source code, combining them later.

We call the "work unit" of the source code a *line . . .*

LINE—

The "work unit" of a source code file.

Eighty or fewer ASCII characters ending with a carriage return.

Enough information to assemble one op code; or to pass a single command to the assembler; or to supply a short comment or a portion of a longer one.

At one time, everyone in the dino computer world knew what a line was, since all messages and all communications were *line oriented.* Should you want to, say, process words, you had to keep each line of characters separate and work with each line individually. Now, this seems incredibly dumb, but that's the way things were. It took the micro people with their memory-mapped video to first see the completely obvious.

But there are a very few jobs remaining where it is a good idea to keep every entry on a separate and unique line that has to stand on its own and has no particular long term relation to the line above or the line below. Assembler source code files are one place where working line by line still is a pretty fair way of doing things.

Quaint but fair.

If you decide to use a "new way" word processor, you will pick up "free form" or full-screen entry where you can see lots of lines at once and easily edit across line boundaries. Great stuff. But, you will still have to keep your own head "line oriented" while you do this.

The lines in the source code are often numbered from one to N, in sequential order . . .

Each source code line is usually numbered in decimal.

The numbers normally start with one and count "by ones," in sequential order, up to "N."

"N" is the number of lines you need to complete the job that the source code is trying to do.

The reason for this numbering is that we need a way to talk about or work with a single line. Instead of saying "the line with the LDA #$56 command," or "the line just above the mustard stain," we say "line number 145." Since the lines are all numbered, we can find line number 145 and work with it. More importantly, so can the assembler.

Actually, your line numbers do *not* normally go inside your source code. It is kinda dumb to waste disk space on things that are easily calculated and not particularly permanent. Instead, line numbers are an *artifact* of the editor or assembler you use. This convenient artifact is normally generated for you by counting the source code lines as they come off the disk or out of RAM and then numbering them on the way to the screen or a printer.

This type of line numbering is very obvious, but it may be different from other computer numbering schemes that you might be familiar with. As some counter examples, machine language programs are located by addresses, and do not use line numbers. BASIC programs use line numbers, but you usually skip around, counting by tens or whatever, and those lines do not have to be entered in sequential order. Pascal does not use line numbers. Instead, the relative position of the line in the program conveys what the line is and what it does.

But, none of these are assemblers. Assemblers normally have line numbers ranging from one to N, in order, with nothing missing and no duplicates. EDASM uses this "one-to-N" scheme. Other assemblers might start their numbering with 1000 or 10000 to keep the number of printed digits constant. These other assemblers sometimes count by tens instead of ones.

One confusing thing about source code file line numbers is that they don't stay unique . . .

> **What was line 145 in one version of a source code might become line 137, or line 193, or might be just plain missing, in a later version of the same program.**
>
> **As the program length changes, or as corrections are made, each line number may point to a different source code file line.**

So, all versions of all source codes are usually numbered sequentially from one to N, counting up "by ones." No missing line numbers are allowed, nor are you allowed to put any line numbers in the wrong order. If you make the source code shorter by deleting something, all the line numbers higher than the deletion *decrease* in value. If you make the source code longer by adding something in the middle, then all the line numbers above the addition *increase by* the amount needed.

Once again, there really are no line numbers in most source code. The line numbers are an artifact generated by the editor or the assembler for your convenience. Line numbers are calculated by counting carriage returns on the end of source code lines as they come off the disk or out of memory.

Thus . . .

> **Regardless of the version, and independent of the meaning of any particular line . . .**
>
> MOST VERSIONS OF MOST SOURCE CODES ARE USUALLY NUMBERED FROM 1 TO N WITH NOTHING SKIPPED AND NOTHING OUT OF ORDER!

The line numbering is usually fully automatic and is done free for you by the editor or assembler. All you have to do is make sure you really mean "line 143" when you say "line 143," because any change in the source code may change the line number.

Two nasty examples.

Say you write a source code and then tell the editor to delete line six, then line eight, and then line ten. What you really did was delete lines number *six, nine,* and *twelve,* because the first deletion bumped everything above line six down a line, and the second one bumped everything above line nine down yet another line.

Or, say you get lazy or in a hurry and don't do a printer dump of each and every version of your source code as you go along. Say further that you add some innocuous line such as some extra white space some place inside your next-to-latest source code version. Now you decide the carry needs cleared. You shove the CLC line in, but what happens? Instead of being where you thought you were, you are one line off, inserting the carry one place beyond where you expected it to go.

More details on this later. One sneaky way to minimize line numbering problems is to *always edit from the high numbers down, rather than from the low numbers up.* That way you are finished with the line numbers that are going to change before they do in fact change.

For now . . .

> **You must keep EXACT track of the line numbers by yourself!**
>
> Line numbers may become wrong if you add or remove any lines from your source code, or if you are using the out-of-date printout from an older version.

Later in chapter five, we'll see an automatic line number changer using WPL that adds, removes, or updates line numbers from word processed source code.

Other assemblers may have different numbering rules or use options. Always check.

OK. So what goes on a line? We already know that a line is the work unit of a source code file, and that a line is some number of characters that will fit neatly across a page or screen that ends with a carriage return.

There are different tasks that each part of a line is intended to do. Because of this, we split the line up into separate areas, each of which has an intended use. These special areas are called *fields . . .*

> **FIELD—**
>
> **A part of a source code line that has an intended use.**

We already know about one field that is called the *line number field*. The line number goes in the line number field. The purpose of this field is to give us or the assembler a way to refer to a particular line. We know the use rules for this field. The line numbers usually will all be in sequential order from one to N, with nobody missing, nobody out of order, and nobody duplicated. Only numbers are allowed in this field. Letters or punctuation are no-no's.

We also know that the line number field is handled more or less automatically for us, so we don't have to worry about it too much. Just be sure that the line number you say is the line number you really mean.
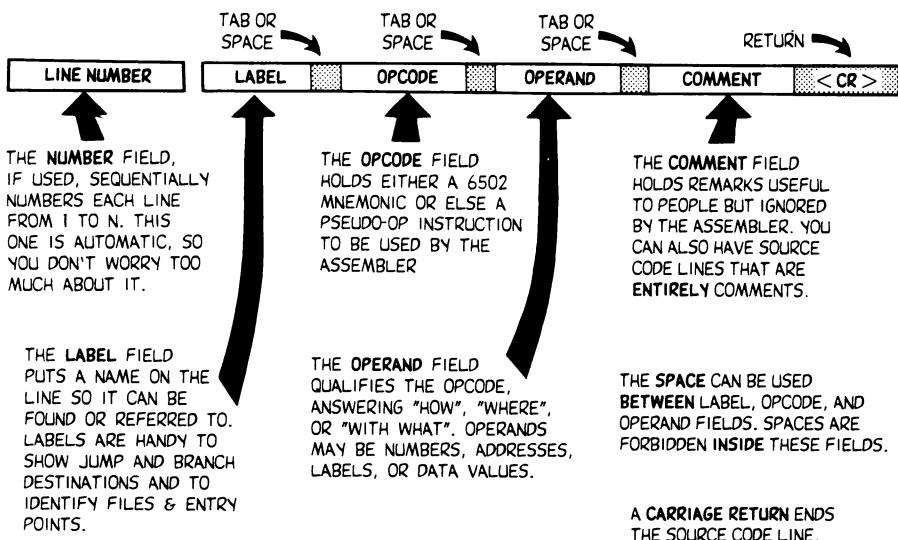
We obviously need more fields than the line number field. In EDASM, there are at least four other fields called, in order from left to right, the *label* field, the *op code* field, the *operand* field, and finally, the *comment* field . . .

> **EDASM usually uses four fields in addition to the "free" line number field.**
>
> **These are called the *label* field, the *op code* field, the *operand* field, and the *comment* field.**

Here's a picture that says the same thing . . .

FIELDS INSIDE A SOURCE CODE LINE

THE **NUMBER** FIELD, IF USED, SEQUENTIALLY NUMBERS EACH LINE FROM 1 TO N. THIS ONE IS AUTOMATIC, SO YOU DON'T WORRY TOO MUCH ABOUT IT.

THE **LABEL** FIELD PUTS A NAME ON THE LINE SO IT CAN BE FOUND OR REFERRED TO. LABELS ARE HANDY TO SHOW JUMP AND BRANCH DESTINATIONS AND TO IDENTIFY FILES & ENTRY POINTS.

THE **OPCODE** FIELD HOLDS EITHER A 6502 MNEMONIC OR ELSE A PSEUDO-OP INSTRUCTION TO BE USED BY THE ASSEMBLER

THE **OPERAND** FIELD QUALIFIES THE OPCODE, ANSWERING "HOW", "WHERE", OR "WITH WHAT". OPERANDS MAY BE NUMBERS, ADDRESSES, LABELS, OR DATA VALUES.

THE **COMMENT** FIELD HOLDS REMARKS USEFUL TO PEOPLE BUT IGNORED BY THE ASSEMBLER. YOU CAN ALSO HAVE SOURCE CODE LINES THAT ARE ENTIRELY COMMENTS.

THE **SPACE** CAN BE USED BETWEEN LABEL, OPCODE, AND OPERAND FIELDS. SPACES ARE FORBIDDEN INSIDE THESE FIELDS.

A **CARRIAGE RETURN** ENDS THE SOURCE CODE LINE.

More details on what each field does will follow shortly. But, if each field is to do some unique job, we obviously need a way to get between the fields and we will need a way to tell which field is where. EDASM uses the "space" character to *tab* between fields . . .

EDASM traditionaly uses the spacebar to "tab" between fields.

Everything up to the first space goes into the label field.

Everything between the first and second spaces goes into the op code field.

Everything between the second and third spaces goes into the operand field.

Everything beyond the third space goes into the comment field.

Since the spacebar is used to get between fields, spaces are not allowed in any of the first three fields. You must use spaces to get between fields. You must not allow spaces inside any of the first three fields . . .

You MUST NOT use any spaces inside the label, op code, or operand fields!

You MUST use a space any time you want to go on to the next field!

After you get to the final, or comment field, you can use any number of spaces any way you want to. But, spaces are strictly a no-no in the label, op code, or operand fields.

You might logically ask, "Why not use a tab command to tab, instead of space?" Well, tabbing was tricky on older Apples, and faking a tab by padding spaces gobbles up space on disk or in RAM. Further, allowing spaces in labels or op codes would create all sorts of other problems. Besides, even on a IIe, the spacebar is much larger and easier to find and use than the tab key.

On "new" EDASM, you can use the IIe tab key to tab if you want to. In fact, this eases "new way" editing by quite a bit. Note that using a tab command to tab is awkward on pre-IIe Apples.

So, the older EDASM rule is that, following the carriage return, the first three spacebar hits force tabs. After that, spaces get used as spaces.

Not all fields are needed on every line. But . . .

Each source code line MUST have something in the op code field.

"Something" is either a command for the assembler or else a real op code for the computer.

So, the EDASM rules so far say that you must have something in the

op code field. Thus, you are not allowed to have a "blank" or "empty" source code line. You can fake blank or nearly blank lines by special commands or by special use of the comment line, as we will see shortly.

Each and every assembler source code line *must* have something valid and useful in the op code field. For any given line, the use of the label field, the operand field, and the comment field is optional. You use these only if you want to or need to.

Let's look at these four fields in more detail . . .

The Label Field

The label field is used to hold, of all things, a *label* that points to this particular line. A label is always defined or equated in the label field.

A label is a collection of letters, decimal points, and numbers that means something. You can use a label to serve as a memory jogger as to what a particular line is for. But the more important uses of labels give the assembler ways to find this particular line, even if the whole program changes size or gets moved around in memory.

For instance, you could use a label called "REPEAT." At the end of your program, you could use an op code of JMP and an operand of REPEAT. When the assembler gets to the JMP command, it finds out where REPEAT really is in memory, and then automatically adjusts the machine language code to get back to the code held in the line called REPEAT.

As a second use of a label, you can name the first line of a subroutine something meaningful. A noise making subroutine could have a label field that is labeled "BEEP." Elsewhere in the program, a JSR BEEP will pick up this sub. The assembler will automatically figure out where BEEP is, and patch in the right code in the right way so the final machine language program runs correctly. Similarly, you might use STALL or WAIT as the name for a time delay subroutine.

Labels are also useful for relative branches. With labels, there is no need to count blocks or do twos-complement arithmetic to calculate branches. Just say BEQ ZORCH, and the assembler will find the label ZORCH and figure out the magic branch value to hit it.

Another important use of the label field is to pre- or post-define an address or a value into something more meaningful. For instance, reading or bit testing location $C030 in the Apple whaps the speaker cone. Early in your program, you can *equate* a label called SPKR to $C030. From then on, anytime anything refers to SPKR, the assembler knows you really mean $C030 and makes the substitution for you. Labels such as HIRES, LORES, PAGE1, PAGE2, TEXT, and so on are much more meaningful and easier to understand than their actual locations.

Labels are also most handy for pointing to files or file entries as well.

Summing up . . .

USE LABELS TO
Jog the programmer's memory Show a jump where to go Show a branch where to go Indicate a subroutine's start Equate an address or a value Name a file or file entry

There are many other uses of labels, but these are the more obvious ones. Study the ripoff modules for other examples of label use.

You do not normally put a label on each and every line in a source code listing. You only provide labels when you want you or the assembler to be able to refer to a line, or to define or equate a label. On the average, only one line out of six or so will have a label in the label field. This, of course, varies all over the lot with your program style and what you are trying to do.

Label fields can be used *before* your main program code to equate values or addresses; or *inside* your main program code to point to a code line; or *after* your main program code to define a file or file value.

If you do not want to use a label, you simply start the line with a space or tab. This will automatically tab you over to the op code field and no label will be used on that line.

It is very important to always use a beginning space or tab on those lines without labels. If you miss this detail, labels and op codes will get all mixed up, plowing the works.

The rules say that a label *must* start with a letter and *must not* have any spaces. It's good practice to limit the length of most labels to five, six, or seven characters, and to use only capital letters, decimal points, and numerals, although most assemblers allow some other symbols as well.

Your choice of letters and numbers should be meaningful and easy to read. Naturally, you'll want to avoid mixing "ones," "eyes," "els," "ohs," and "zeros." It is a long tradition among programmers to use very creative label names that are funky and far out. But, you should do this only to the point where it still aids, rather than confuses, someone else who is trying to understand what is happening.

With most assemblers, a label can only be defined at one place in the source code, and thus is only allowed to appear on *one* label field in the entire source code. That same label name can be used many other places in the same source code to *refer* to that label field, but there can be no doubt as to where the label is pointing.

Should you need two or more labels that do pretty much the same thing, change each label slightly by adding numbers to the end or else use creative misspelling. Assemblers with a local label capability will let you reuse labels, by following their use rules.

Let's sum up the label field rules . . .

LABEL FIELD RULES

A label can only be defined in ONE
label field in the source code, but any
label can be referred to as often as is
needed elsewhere in the listings.

There usually must be a label field
somewhere in the source code for
each label reference. The thing you
are referring to must exist.

Labels must start with a letter but can
be a mix of letters or numbers. No
spaces are allowed.

All labels must be unique.

So, you use the label field of a source code line only when you want
to refer to that line, either for use by the assembler in jumps,
branches, subroutine calls, or equates; or by a programmer or user as
a memory jogger. Each label must be unique and can only be used
once, unless your assembler allows separate global and local labels.
Each label should have a meaningful name and should start with a let-
ter and be a five-to-seven-character mix of only letters, numbers, and
limited punctuation. Should you not want to use a label, you tab over
the label field with a space or tab command.

Upper case labels are pretty much standard.

Three points of possible confusion. First, if you want to define a
label *externally* to a source code module, most assemblers will let you
do this. Just follow the rules.

Secondly, it is legal to name a label and then never refer to it. This is
handy as a memory jogger or to save room for later expansion. You
might get a question mark on these in a label listing. That is just the
assembler's way of making sure you really want this done. But unused
labels are both handy and legal.

Finally, remember that labels can be defined before, during, or after
those source code lines that will actually get assembled into the "real"
machine language instructions. Before *equates* an address or a value.
During *points* to a program line. After *defines* a file or a file entry.

By the way, most assemblers have enough smarts to scan the entire
source code at least once before they begin assembly to pick up all
the labels ahead of time. So, you can define a label anyplace you
want, either before or after the label gets used. It is best practice to
"EQU" or *equate* a label before it is used.

By the way, labels are sometimes called *symbols*. Label tables are
sometimes called symbol tables, and any assembler that uses labels
can also be called a *symbolic assembler*.

Nuff for now on the label field. The next field over is called . . .

The Op Code Field

This field is where all the action is. Remember that each and every source code line must have something useful and legal in the op code field.

There are two things that might go in this field. These are *real op codes,* and *pseudo-ops . . .*

REAL OP CODE—

A three- or four-letter mnemonic that will be assembled into actual machine language code.

Typical examples are LDA and JMP.

PSEUDO-OP—

A three- or four-letter mnemonic that will tell the assembler it is to do something different or special.

Typical examples are ORG and EQU.

The only legal things you can put in the op code field are mnemonics to be assembled or special instruction mnemonics that deliver a message to the assembler. Anything else is a no-no. We will see much more on pseudo-ops shortly.

Note that the field *position* is the only way the assembler has of telling where the op code is to be. If you do a CLC mnemonic and forget the leading space that tabs you into the op code field, the assembler thinks you have a *label* called CLC . . .

Op codes are ONLY allowed to appear in the op code field and labels are only allowed to be equated or defined in the label field.

ANY MIXUP SPELLS TROUBLE!

Anything else spells trouble.

Right here in River City.

Note that only a mnemonic is allowed in the op code field. Now, a few op codes and some pseudo-ops are complete in themselves and need no further information to finish their intended tasks. We call these op codes *implied* instructions, and typical examples are NOP, CLC, TAX, or INY. PAGE is an example of an upcoming pseudo-op that needs no further help or information.

But, for most other op codes and pseudo-ops, we need to answer the questions "With what?" "Where?" or "How much?" That sort of information goes in the next field over and is called . . .

The Operand Field

An *operand* is something that qualifies an op code, giving us further needed information . . .

> **OPERAND—**
>
> A "qualifier" for an op code that gives further information, such as answering "Where?" "What?" or "How much?"
>
> Operands are essential for most op codes or pseudo-ops and are the usual way the assembler can decide which addressing mode is wanted.

The operand field must be used any time there is any doubt as to which addressing mode is intended. The operand can be a value or an address, or even some arithmetic instructions that will lead you to a value or an address. Operands for pseudo-ops can also be things like ASCII text, data values, or other file entries.

Most usefully, operands can also be labels that have been elsewhere defined or equated to values, addresses, or other lines in the program. Thus, while labels are *defined* in the label field, they are normally *used* in the operand field. Labels can be used many times, but are only allowed to be defined once.

It is the operand that sets the address mode, and decides whether we are talking hexadecimal, decimal, binary, ASCII, or even —urp— octal. You must pay extreme attention to detail on your operand, for the slightest mix-up in symbols or usage will foul the works up royally. We will go into more details on operands later in this chapter.

Our final field on the source code line is called . . .

The Comment Field

The comment field contains remarks that are only useful to the programmer or final user . . .

> **COMMENT FIELD—**
>
> Contains remarks or formatting useful to the programmer or user but ignored by the assembler.

The comment field is ignored completely by the assembler. A comment field can also be used to draw fancy boxes, to add white spaces, and to do other pretty printing that makes the source code more readable. You can put lots of spaces or most anything else you want in the comment field. The only exception is that you are only allowed a single carriage return that must go at the end.

You should use your comment field to explain what is happening as it is happening. Where a comment is too long for one line, you can break it up, and pick up the next part of the comment in the comment field of the next line. You might want to indent continuing comments to make them more readable.

The ultimate rule on comments is . . .

> **The ultimate comment rule—**
> ## THE MORE, THE BETTER!

We will see detailed examples of how to use all these fields later. For now, we simply want to find out what each field is used for.

To avoid mix-ups between operands and comments, it pays to start each comment with a space and a semicolon. I like to follow this with another space that makes things more legible. Thus, after your operand, you type a tab or tabbing space, a semicolon, then a genuine space, and then your comments.

There is one special and set-aside use rule of comments that involves the symbols "*" and ";" . . .

Any EDASM line that starts with a "*" or a ";" is treated entirely as a comment.

That entire line is ignored by the assembler, but is otherwise most useful for documentation.

Thus, if you want to put down a title block or some use rules, you start each line with a semicolon or asterisk. I like the semicolon since it is cleaner and less obnoxious, but the asterisk is more traditional. Either one works. Again, we'll see more details when we get to the ripoff modules.

Let's see where we are.

An assembler source code file holds all the information we need to tell an assembler to put together a machine language program for us. The work unit of a source code file is called a line. Lines are often numbered in sequential order from one to N, but the line number itself is not normally part of the source code.

Each line is broken up into fields. You get from one field to the next by tabbing or else using the spacebar as a tab. You must space to get between fields. Spaces inside the label, op code, and operand fields are prohibited.

Besides the "free" number field, there are usually four fields, called the label field, the op code field, the operand field, and the comment field.

Unless you have used the special "*" or ";" comment markers as the first character of your line, each line must have an entry in the op code field. This entry can only be a legal 6502 op code or a legal pseudo-op needed to get the assembler's attention. Thus, you normally must have an op code, but everything else is optional.

The label field gives you or the assembler a way to refer to this line.

The op code field holds a real op code instruction to be assembled into the machine language object code, or else a pseudo-op command to be passed only as far as the assembler program.

The operand field gives any additional information needed by the op code; such as qualifiers that answer "How much?" "With what?" or "Where?" Labels may be used as operands to conveniently point to other lines in the program or to sit in for previously defined or equated values.

Finally, the comment field gets ignored by the assembler, but gives you a good way to insert remarks and notes into the source code file. You can extend the comment field to be the only field on a line by using the "*" or ";" symbols as the first character on a line.

Let's now pick up some more detail on that all-important operand field . . .

MORE ON OPERANDS

It is the operand field that passes on most of the needed information to the assembler program. So, this field is the one that will cause the most problems for you if you are not *extremely* careful.

There are three very important things the operand field does. First, it picks the number base you are using. Secondly, it sets the addressing mode for the mnemonic you used in the op code field. Thirdly, the operand field chooses between labels and fixed values as a "qualifier" for the op code.

Thus . . .

```
+--------------------------------------+
|          THE OPERAND FIELD           |
+--------------------------------------+
|                                      |
|  1. Picks the number base in use.    |
|  2. Selects the address mode.        |
|  3. Chooses labels or fixed values.  |
|                                      |
+--------------------------------------+
```

Because the operand field does so much with so few symbols, you must be extremely careful what you put here . . .

```
+------------------------------------------+
|  You must be EXTREMELY careful when      |
|  writing to the operand field!           |
|                                          |
|  Wrong symbols or illegal values will use|
|  the wrong number base, the wrong        |
|  address mode, or will fail to assemble at|
|  all, giving you an error message.       |
+------------------------------------------+
```

The first thing the operand field needs is a way to tell which number system is in use. Most assembly language programmers use hexadecimal as their main and standard number base, since this is the number base that most easily interacts with the Apple II or IIe on the gut level. In fact, to become a decent assembly programmer, you must eat, live, breathe, and think in hexadecimal all the time.

You can usually tell long-term assembly programmers by the extra six fingers they eventually grow. If you get into hex, you might as well go whole hog.

Decimal numbering is sometimes needed if you are writing an assembly language module that is to interact with BASIC or another high level language. These higher level languages often use a decimal number from 0 to 255 to represent an 8-bit data value, and use 0 to 65535 to show a 16-bit address location or other value.

Anyway, most people use hex for most assembly language programming. To tell a hexadecimal, or base 16 number, EDASM *demands* a "$" dollar sign in front. Decimal numbers are shown without anything in front . . .

> **Hex numbers normally must be shown
> with a "$" in front, such as $F2, $12A6,
> $CAFE, or $23.**
>
> **Decimal numbers are always shown with
> nothing in front, such as 25, 255, 6789,
> or 12345.**
>
> **Use no internal decimal commas.**

Some non-6502 assemblers may instead put an "H" after a hex number, rather than a "$" before. Thus, $CAFE = CAFEH. Virtually all 6502 assemblers use the $ notation.

A very important rule . . .

> **If you forget to use the "$" sign in front
> of a hex value, you will get a "free"
> conversion of decimal to hex, and the
> wrong values will get entered into the
> program . . .**
>
> **DON'T FORGET THE $ SIGN!**

You can also enter octal values into EDASM by using the "@" prefix, but if you do, people will laugh at you, snicker behind your back, and no longer associate with you. They may even kick sand in your face.

Many assemblers, including "new" EDASM, will let you use a "%" prefix for binary values. Binary values are useful when you want to see what each and every line on a port is up to, but use of binary is otherwise very cumbersome. Naturally, you must use eight binary ones or zeros for an 8-bit value, and 16 ones and zeros for a 16-bit value. If you do not have binary on your assembler, just convert to hex before entry.

There are also ways to enter long strings of ASCII characters, pairs of addresses, and single or multiple file values into the operand field. This is done by using the pseudo-ops we will look at shortly.

For now, though, there are usually two kinds of numbers you will put in an operand. You most often will use hexadecimal, being very careful to put a "$" sign in front. Should you want to use a decimal value instead, you enter this as a plain, old, whole number without commas and without anything in front. This decimal value will automatically get converted to hexadecimal as it is assembled into the machine language object code.

Be sure to use decimal values between 0 and 65535. The "inverted decimal" values of −32767 to +32767 common to Integer BASIC must be converted to their positive decimal or else to their hex values if EDASM is to use them. *The Hexadecimal Chronicles* (Sams 21802) makes this a snap.

Some assemblers, other than EDASM, will allow negative decimal values.

The operand, of course, is fulfilling the needs of the op code in front of it. So, the size of the number has to match the needs of the op code . . .

> The size of an operand value must match the needs of the op code it is qualifying.
>
> Use two-digit values from $00 to $FF, or 0 to 255, for 8-bit values such as an immediate load, a page zero address, or a file entry.
>
> Use four-digit values from $0000 through $FFFF, or 0 to 65535, for 16-bit values, such as absolute addresses.

If you want an 8-bit operand, use a value from $00 to $FF. If you instead want a 16-bit operand, use a value from $0000 to $FFFF. Be sure to get the right length operand to go with what the op code wants and needs, or you will get an error message or the wrong entry.

If you use a label as an operand, as you often should, the rule stays the same. Use a label elsewhere defined to 16 bits for 16-bit values, or a label elsewhere defined to eight bits for 8-bit values.

By the way, there is a quirk in the page zero addressing of EDASM that pretty much demands you throw in a zero in front of single-digit numbers. Thus, it is a good idea to use $05 instead of $5, and $050A instead of $50A. Always use 8-bit or 16-bit values, rather than 4-bit or 12-bit ones.

OK. To show a hex number, we put a dollar sign in front. To show a decimal number, we put nothing in front. You must make the size of the operand qualifier match the needs of the op code, using an 8-bit or a 16-bit value or address when and where needed.

Our next hassle is deciding . . .

Which Address Mode?

The operand also sets the address mode for us. This is done by showing the operand how many bits the value is, and by including special symbols that include "#" for immediate, "( )" for indirect, and "," for indexed addressing modes.

Let's quickly review the 6502 address modes and see what notation you have to use with which mode . . .

IMPLIED ADDRESSING

Implied addressing needs no more information to carry out a task.

Implied operands are usually blank, such as . . .

NOP
TXA
CLC
PHP

ACCUMULATOR ADDRESSING

Accumulator addressing is really implied addressing that applies only to the accumulator.

While no more operand information is needed, EDASM *demands* that you use an extra "A," like so . . .

```
ROL  A
ASL  A
ROR  A
LSR  A
```

Three quick notes here. First, there are only four accumulator-mode 6502 mnemonics. Secondly, while no further information is needed to handle an accumulator-mode instruction in the Apple, EDASM apparently needs some way to tell an accumulator rotate from a rotate or shift in some other mode. Thus, you *must* provide an extra "A." On "new" EDASM, this "A" *must* go in the operand field, and *must* be preceded by a space.

Thirdly, the label "A" is reserved by EDASM and you must not use it for anything else. You are allowed any other single-letter labels, except for "X" or "Y." These single-letter labels turn out most useful.

Onward and upward . . .

IMMEDIATE ADDRESSING

Immediate addressing puts a value into a register. The immediate mode needs an 8-bit value as an operand.

Immediate operands MUST begin with a "#" symbol . . .

```
LDA  #$60
LDA  #60
LDA  #ZORCH
```

You absolutely *must* have the "#" symbol in front of any immediate operand. The "#" is the *only* way EDASM has to tell an immediate address from the upcoming page zero addressing.

In the first example, we fill the accumulator with the value hex $60. In the second example, we fill the accumulator with the value hex $3C, since the decimal value of 60 equals hex $3C.

Watch for that missing "$"!

Note that the "#" sign always goes first, saying "the *immediate* value." This is followed by the "$" sign that says "in hexadecimal." Together, the two symbols say "the immediate value in hexadecimal." The sharp *always* goes first!

Watch this key detail. Sharps first, dollars later. A musician gets paid *after* he plays.

In the LDA #ZORCH example, EDASM will try to use the label ZORCH as an immediate value. It can do this only if ZORCH has been previously equated or defined in the program, and then only if ZORCH is an 8-bit number ranging from hex $00 to $FF.

Two key points . . .

> **The first symbol in an immediate operand MUST be a "#."**
>
> **If you are using a label, that label MUST have been defined or equated elsewhere in the program, and MUST be an 8-bit value ranging from hex $00 to $FF or going from decimal 0 to 255.**

The absolute short addressing mode of the 6502 goes by the name of . . .

> **PAGE ZERO ADDRESSING**
>
> Page zero addressing refers to an 8-bit address on page zero. Values are loaded from or stored into that page zero address.
>
> Page zero operands consist only of an 8-bit address value . . .
>
> ```
> LDA   $06
> BIT   93
> STY   SNARF
> ```

Once again, be sure to use that "$" dollar sign in front of all hex numbers. Note that there is no "#" sharp symbol used for page zero addresses.

The missing "#" is how EDASM tells a page zero address from an immediate value. For instance, LDA #$06 puts the number "six" into the accumulator. A LDA $06 goes down into page zero, and takes whatever is in location $0006 and puts that value into the accumulator. The *value* in $0006, of course, can be any old 8-bitter ranging from $00 to $FF.

In our BIT 93 example, we BIT test address space location $5D. Why $5D?

The real power of a full assembler like EDASM lies in its use of labels, But, once again, if you use a label like SNARF, you have to equate or define SNARF elsewhere in the program. In this case, SNARF must be an 8-bit address between hex $00 and $FF, or decimal 0 to 255, that is intended to point somewhere in the address space locations of $0000 through $00FF.

A gotcha . . .

> **EDASM insists that you EQU all page zero address labels ahead of time.**

It is always best practice to equate a label *before* you use it, rather than after. But, EDASM or any other good assembler will find the label anywhere in the source code.

Another mode . . .

> ## ABSOLUTE ADDRESSING
>
> Absolute addressing refers to some 16-bit address somewhere in the entire address space.
>
> The operand has to be a 16-bit value . . .
>
> ```
>         JSR   $C67E
>         STY   65521
>         INC   WIMP
> ```

Absolute addressing is one of the most popular among beginning programmers. It can hit any location in the entire address space. Once again, the hex values start with a dollar sign, and a label like WIMP must have been equated or defined elsewhere in the source code. This time, the label has to be a 16-bit address value from $0000 through $FFFF, or decimal 0 through 65535.

Moving right along, branches and tests always involve themselves with . . .

> ## RELATIVE ADDRESSING
>
> The relative addressing mode is used with tests and branches to go so many steps forward or backward in memory from the present address.
>
> The 6502 lets you go forward or backward 127 locations, using an 8-bit twos complement signed binary number.
>
> The operand for a relative address is almost always a label, pointing to where the branch is to go *to* if the branch is taken . . .
>
> ```
>         BEQ   RETURN
>         BCS   NXTVAL
> ```

In the BEQ RETURN example, if the zero flag is not set, the program continues on to the next instruction. If the zero flag is set, the program branches to the line that has the label RETURN on it. The assembler will automatically figure out the branch value for you, so long as it is legally within plus or minus 127 addresses from where you are. Similarly, the second example will go to NXTVAL if the carry flag is set, and will continue on in sequence if the carry flag is cleared.

One of the great things about using labels is that the assembler can automatically figure out how many squares forward or backward to go on a relative branch, even if the program gets longer or shorter. Thus, while you could do a BEQ $3F14 or something similar, this ties you down badly, and it is far better to use a label . . .

```
ALWAYS use labels for relative address
operands.

Fixed relative address operands will
return to haunt you on any program
changes or relocations.
```

The label must, of course, be defined elsewhere in a line field and must point to a source code line that is within plus or minus decimal 127 addresses of where you presently are.

By the way, "new" EDASM now includes an automatic printout of each "branch taken" destination address. This is very useful for analysis or troubleshooting.

Here's a fairly fancy address mode . . .

```
                INDEXED ADDRESSING

Indexed addressing goes to the sum of
a base address plus a register value,
and then works with that address.

The 6502 offers page zero indexed by
X or Y and absolute indexed by X or Y
as "pure" indexed modes. Use of
page zero indexed by Y is limited.

The operand must be an address of
the correct length, followed by a
comma, followed by the name of the
index register . . .

        LDX    $03,Y
        ORA    126,X
        SBC    $8034,X
        CMP    34126,Y
        LDA    GREEBLE,X
```

In the indexed modes, you need an address, followed by a comma, followed by the name of the index register being used. Note that GREEBLE,X can refer to a page zero indexed mode if GREEBLE has

been previously defined to be an 8-bit number. GREEBLE will refer to a 16-bit absolute address if GREEBLE has been previously defined as a 16-bit address from $0000 to $FFFF. The assembler will figure out the address mode for you, but don't confuse it, or it is likely to return the wrong op code.

Note also that there are only two op codes available that use "page zero,Y" addressing. These are LDX SNORK,Y and STX SNORK,Y. These are two of the four "cross-indexed" commands in the 6502. Cross-indexed commands are my favorites, since there is so much you can do with them in so many sneaky ways.

When you want to go to one address to get a second address, you can use . . .

INDIRECT ADDRESSING

The indirect addressing mode goes to one address pair to find the final address it is to go to.

On older 6502's, only the JMP command is available as a "pure" indirect command.

The operand uses parentheses for an indirect command . . .

JMP ($6CA7)
JMP (2786)
JMP (PART2)

Any time you need an indirect command that goes to one address to get a second one, you use parentheses in the operand. The parentheses mean "go to this address and the one immediately following to get the 16-bit address you really want. Then use that address."

When you use a label here, that label must be elsewhere equated or defined as an absolute address. As with relative branches, it is nearly always a good idea to use labels for indirect addresses.

The "pure" indirect addressing mode is limited on the 6502 to the JMP command. But note that you can fake a JSR indirect by doing a JSR to JMP indirect. This takes a few extra bytes and a few extra machine cycles, but it works. Other indirect actions are handled with the next addressing mode.

A bug exists in older 6502's that prevents indirect addressing from crossing page boundaries. Do not use indirect addressing on the top byte of any page. In fact, it's a good idea to avoid pure indirect entirely. This bug is fixed on the 65C02's, which have bunches of useful and properly working indirect commands.

The real heavy of the 6502 world combines indirect addressing with indexed addressing . . .

INDIRECT INDEXED ADDRESSING

This very popular and often-used 6502 addressing mode finds an address pair on page zero and then adds a Y index value to it to calculate the final working address.

The indirect indexed addressing mode is most useful for 16-bit addressing and working through long files.

The operand is a page zero address in parentheses, followed by a comma, followed by a Y . . .

```
LDA    ($14),Y
STA    (65),Y
CMP    (BIGFILE),Y
```

This is sort of a "two-for-one" deal, where you combine indirect addressing and post indexing that takes you "Y" locations beyond the indirect address.

Here are two important rules on how to use this super powerful address mode. First, you must use a page zero address inside the parentheses. If you use a label here, it must be only an 8-bit value, and must, of course, be equated some other place in the source code. Secondly, note that you must use the Y register as an index value. There is no mode of LDA ($88),X available on the 6502.

As a handy hint, the indirect indexed address mode defaults to "pure" indirect addressing if you force a Y index value of zero. This gives you an easy way to fake a "pure" indirect ADC, AND, CMP, EOR, LDA, ORA, SBC, or STA.

The indirect indexed addressing mode solves the hassle of going beyond 256 locations in a file or a data bank. By changing the page zero base address inside the parentheses, you can reach anywhere in the 6502's entire 65536 location address space. Another handy use of this mode is to use one block of code to access several different files or data blocks, just by changing the indirect base address values or *their* label names.

It's off to oddball city for a check into our final 6502 address mode . . .

INDEXED INDIRECT ADDRESSING

A seldom-used addressing mode that goes to the sum of a page zero base address plus an X index, gets a full 16-bit address from that location and the next one, and then uses that final calculated address.

Occasionally is used to pick one of a group of possible addresses that are stashed on page zero.

The operand is in parentheses. Inside the parentheses is the page zero base address, a comma, and an X . . .

```
LDA  (05,X)
LDA  (IRQLOC,X)
```

This one is an oddball and doesn't get used much since it gobbles up too much valuable real estate on page zero. If you force X = 0, then this addressing mode will also act as a "pure" indirect mode. Such use is rare.

Try not to confuse this *pre-indexed* addressing mode with the very useful and most handy *post-indexed* mode called indirect indexed. Note that the "pre" or the "post" points to where the index is in the name.

"Pre" is obscure and seldom used. "Post" is the heavy.

Indexed indirect is rare; indirect indexed is well done.

But if you do use pre-indexed mode, be sure you have a table of address pairs on page zero, and that your X pointer gets doubled so it increments *by twos,* since you have to point to one address *pair* at a time.

Let's sum up the operand notation you use for each address mode . . .

OPERAND SUMMARY		
Mode	**Symbol**	**Examples**
IMPLIED		CLC
		NOP
ACCUMULATOR	A	LSR A
		ASL A
IMMEDIATE	#nn	LDA #$35
		LDA #VALUE
PAGE ZERO	nn	LDA $06
		LDA PLACE
ABSOLUTE	nnnn	LDA $2A14
		LDA PLACE
RELATIVE	a label	BEQ PLACE
		BVS NEWSTUF
INDEXED	nn,X	LDA $02,X
	nn,Y	LDX PLACE,Y
	nnnn,X	LDA $23C4,X
	nnnn,Y	LDA PLACE,Y
INDIRECT	( )	JMP ($0216)
		JMP (PLACE)
INDIRECT,Y	(nn),Y	LDA ($17),Y
		LDA (PLACE),Y
PRE-INDEXED	(nn,X)	LDA ($19,X)
		LDA (PLACE,X)

Use $ in front of all hex values.
Use nothing in front of decimal.
nn = 8 bits, nnnn = 16 bits

Once again, that final pre-indexed addressing mode is rarely used. When picking an address mode, use nothing for implied, an "A" for accumulator, a "#" for immediate, two hex digits for page zero, four hex digits for absolute, a label for relative that is within 127 slots forward or backward from where you are, a set of parentheses for indirect, a comma and a register name for indexed, and a set of parentheses holding a page zero address followed by a comma and a Y for indirect indexed.

One obvious rule . . .

Not every 6502 address mode is available for any particular 6502 instruction mnemonic, so . . .

THE OPERAND MUST MATCH THE OP-CODE!

This says that if you try to use any 6502 instruction in an illegal mode, the assembler will stop and generate an error message. Always

check your pocket card or 6502 plastic card to make sure an address mode is legal before you try to use it.

Some of the pseudo-ops will also need operands. The operand might be an address, a definition, an ASCII string, a group of data values, or a special location. We will see the individual uses of these special pseudo-op operands as their needs come up.

As an aside, the newer 65C02 has several more "legal" addresses modes and a near-infinite supply of "illegal" ones, just waiting for your use. If you use the 65C02, make sure your assembler can handle these new mind-blowing 65C02 op codes. "New" EDASM does allow these new 65C02 op codes. An add-on 65C02 module is also available for the S-C Assembler. Naturally, your final object code will only run on a machine with a 65C02 in it. Certain legal 65C02 commands will hang an older 6502.

Operand Arithmetic

One very interesting and very heavy feature of EDASM is that you can do simple arithmetic on operands.

For instance, you can do a STA PLACE+1; or a LDA ADDRESS/256; or a STA OLDFLE+FLNGTH; or an ORG START+$0200 or a STY BASE+POINTER*2 . . .

You can do arithmetic as part of an operand with EDASM.

This lets you calculate an address or value. The calculation may be either absolute or relative.

You can add, subtract, multiply, or divide, using the usual +, −, *, and / symbols. Use no spaces.

This gets very convenient for skipping part of memory when you assemble a new part of a program, for picking two parts of an address pair, for doubling a pointer value, and for other sneaky things.

There are some use rules and restrictions to operand arithmetic. You must use only the digits from 0 to 65535 or hex $0000 to $FFFF, or labels equated to values in this range. If you are working with 8-bit values, you have to be sure you get an 8-bit result. On 16-bit values, the answer will wraparound, giving you a 16-bit result.

And, you are not allowed to have any spaces in the math expression. Anything beyond a space is usually treated as a comment.

So . . .

If you use operand arithmetic, be sure to have no spaces in the expression.

Also, be sure that you get an 8-bit result when you need one.

The best way to work with operand arithmetic is on a "try-it-and-see" basis. If it works, fine. If not, you'll get an error message that may suggest a better approach.

Operand arithmetic is a sophisticated tool for the advanced programmer. You don't have to use it at all if you don't want to. But operand math sure does neat things when you really get into it. We will see some examples later.

Logic functions are also allowed in "new" EDASM, using "~" for AND, "|" for OR, and "!" for EXCLUSIVE OR.

MORE ON PSEUDO-OPS

We have seen that two different beasties can go in the op code column. We can use "real" op codes that are to be assembled into 6502 machine language code. And we can use "fake" op codes that deliver a message to the assembler, making it change what it is doing . . .

> **A pseudo-op delivers a message to the assembler during the assembly.**

Thus, pseudo-ops are used to control the assembler, while real op codes are used by the assembler to generate final 6502 machine language object code.

When you write a source code line, you have a choice of making that line a pure comment, of giving a real op code that you want to assemble into a machine language program, or of sending a special message to the assembler for use during the assembly process.

What you put where is up to you. This will become more obvious as we go on.

In this book, we will classify the EDASM pseudo-ops into four groups. Further, if the pseudo-op is very "important" or is very often used, you will see it described in a separate section. There are lots of pseudo-ops that you won't have to worry about too much until well after you have some assembler experience. We will call these "also rans," and we will just summarize them for now. Feel free to tear into the EDASM assembler manual for more details on these.

Additional pseudo-ops unique to either version of "new" EDASM are summarized in Appendix A.

Let's call our pseudo-op groups the *pretty printing* group, the *structure* group, the *file* group, and finally, the *conditional* group . . .

> ### TYPES OF PSEUDO-OPS
>
> The PRETTY PRINTING pseudo-ops make the final assembler easier to read and understand.
>
> The STRUCTURE pseudo-ops decide where our program is to start, isolate file blocks, and chain listings together.
>
> The FILE pseudo-ops help you enter data values into a file.
>
> The CONDITIONAL pseudo-ops give you a way to assemble logically selected parts of a complete listing.

Let's start with the pretty printers . . .

Pretty Printers

Our pretty printing pseudo-ops make the assembler listing easier to read and easier to use.

So many assembly language listings look so awful because the people doing them are either lazy or else afraid to "waste" some source code space on commands that make the assembly listing much more attractive and far easier to read. As we'll find out later, well over one half of your source code should be devoted to user comments and to pseudo-op commands that make the listing pleasant and readable.

SKP

We will start with a very simple pseudo-op. SKP stands for *skip* and lets you skip as many vertical spaces as you want to on your assembler printout. For instance, SKP 1 will give you one blank line. SKP 6 will give you a white space six blank lines high on the printout.

You use SKP to put white space into the printout. This lets you separate sections, makes titles more attractive, and generally cleans up the presentation. Remember that a blank source code line is a no-no. So, if you want a blank line, use SKP 1 instead. This is sort of like some military documents with their " This page intentionally left blank" message on them, but that's the way it is.

One use rule. Be sure to leave a space between SKP and the number. It's SKP 5, not SKP5.

PAGE

The PAGE pseudo-op does a form feed for you. A form feed automatically moves you to the top of the next page. Use this one to put the listing breaks where you want them and to isolate separate parts of a program, such as the title block, the hooks, the main program, the subs, and the files from each other.

The "old" version of EDASM has a bug that sometimes messes up headers if it is allowed to do its own page breaks. So, it is always a good idea to force PAGE commands when and where you want them.

SBTL

The SBTL pseudo-op will print out a *subtitle* for you at the top of each page and will automatically number the pages for you with a page number, a version number, and the name of the programmer. The SBTL command reads a disk file called the ASMIDSTAMP, short for "assembler identity stamp." You are prompted when you first boot EDASM to update this header.

It is a very good idea to always use this SBTL pseudo-op as the first line in your source code file. If you do not use SBTL, you will get no headings and no page numbers.

LST OFF

This pseudo-op says to turn the listing off. You do this if you don't want to have the entire assembly process appear on your printed record. If you are reassembling only a small part of your entire program, you put a LST OFF at the beginning, and then the LST ON command at the start of where you want a printed record.

I do not recommend using LST OFF this way, since you don't always have a complete update of the last version of your assembly source code. Leave the listing on during assembly.

One handy use of LST OFF, though, is to put this at the end of your program, after you have gone through it a time or two. This stops the printing of the long label lists at the end of your program and speeds things up considerably.

LST ON

This is the opposite of LST OFF and turns the printer back on when you get to a point in the program when you want to start listing again. EDASM automatically comes up with the listing on, should you give it the proper print command.

On "new" EDASM, many different listing features can be individually switched on or off. See Appendix A for more info on this.

ALSO RANS—

There are two older, lesser pretty printing pseudo-ops. These are *REP* for *repeat* and *CHR* for *character*. Together these two let you print a continuous line of stars or whatever.

It is simpler and easier just to use comments for the same thing although using REP and CHR does save a negligible amount of source code disk space. These commands are not needed at all if you are creating your source code the "new way" with a word processor. The auto-repeat on the Apple IIe completely eliminates the need for either of these.

See Appendix A for additional "new" EDASM pretty printers.

Structure Pseudo-Ops

There are three very important pseudo-ops used to structure your source code, along with seven lesser ones.

ORG

The ORG pseudo-op tells us where we are to start addressing our object code program. Every normal machine language program needs

a starting point, and your source code file must tell the assembler where to start. This must be done before the assembly into machine language can begin.

For instance, an ORG $0800 early in your source code will tell the assembler to start assembling at $0800.

We'll see later why you usually put the origin on the third line of your source file. You must put it before any real op codes.

You can also use ORG later in your program to reposition a new part of your code. Say you want your subroutines to begin on the page above your main program. You can use an "ORG START + $0100" command to do this. The "ORG START + $0100" tells the assembler, "Forget about where we are assembling. Instead, start assembling what follows at the address START + $0100." Note that this operand does a 16-bit address calculation. For this to work, you must have a label of START on your first "real" program line.

Now, if your main program was longer than a single page, you would use START + $0200 or START + $0B00, or whatever you wanted.

ORG can also be used to generate several different object codes from a single source code file. More on this later.

EQU
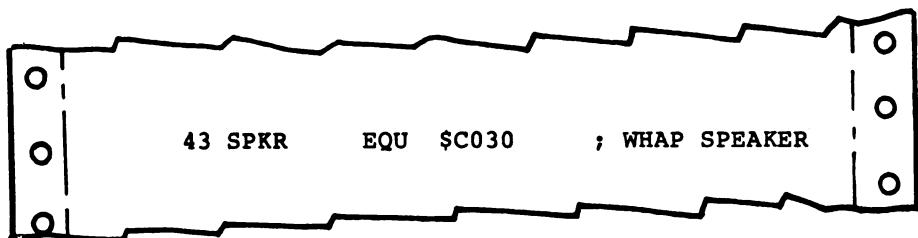
The pseudo-op EQU stands for *equate,* and is used to first establish the meaning of a label or a page zero address, an absolute address, or a 16-bit value. You must be very careful to use 8-bit values for 8-bit equates and 16-bit values for 16-bit equates . . .

Use 8-bit values for 8-bit EQUs.
Use 16-bit values for 16-bit EQUs.

DON'T MIX THEM UP!

All the EQUs should go ahead of the real op codes so that all labels are defined before they are used.

For instance, a line of . . .

```
43 SPKR      EQU   $C030      ; WHAP SPEAKER
```
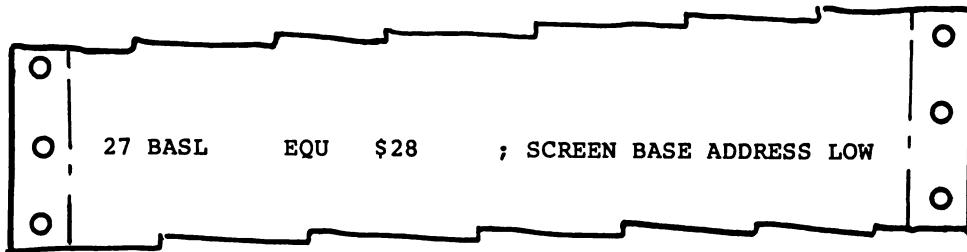
. . . will define the label SPKR as having an address of $C030. Any time later in the program that you want to whap the speaker, you just use SPKR as an operand.

Note the separate fields above in our source code line. The number field tells us we are on line 43. The SPKR in the label field defines this label for us. Since we have defined this label here, we are now free to use SPKR as an operand as often as we want to later in the program. The EQU in the op code field tells us we are using a pseudo-op that

instructs the assembler to do something, rather than a "real" op code that would get assembled into object code. EQU means "equate," so SPKR will get equated to whatever is in the operand field.

The operand field has a $C030 in it, meaning that we will equate SPKR to this hexadecimal value. Finally, our comment field tells the programmer or anyone else what this line is up to and why it is being used.

You also use EQU to set up a page zero address. For instance, this line . . .

```
27 BASL      EQU     $28       ; SCREEN BASE ADDRESS LOW
```

. . . tells us that BASL really means page zero address $28, any time we use this label as an operand.

These EQUs are very important, and you usually equate everything near the beginning of the program, long before you use any of these labels.

Every label must be defined once and only once in some label field. There are at least three ways to do this: (1) you can use the EQU pseudo-op to define a label as an address or a value ahead of time; or (2) you can use a label in a label field of a source code line to define a label as a working address pointing to a "real" program line in your object code; or (3) you can use the upcoming DFB pseudo-op to define a label as a file start or a file entry.

We'll have more to say on EQUs later.

CHN

This one stands for *chain*, and gives us a way to tie two or more source code files together. For instance, a command of CHN PART2 will go into the disk and continue with part two of the assembly. CHN should always be on the *last* line of any source code file, since that file gets replaced with a new one on the chain command.

When you chain two programs together, all the labels and the current assembly address are passed on to the second source code module. CHN is somewhat similar to the "continue printing" command in a word processor program.

ALSO RANS—

There are six lesser used structure pseudo-ops.

OBJ doesn't do anything in "old" EDASM. In "new" EDASM, OBJ lets you do in-place assembly. The OBJ pseudo-op also lets you assemble the code in the Apple's memory in a place different from where it is to actually run. "OBJ" stands for *"object code location"*; compared to "ORG" which shows the final starting place where the code is to actually run.

DSECT and *DEND* set aside an area of memory without forcing you to put actual values into each location. These pseudo-ops are not often used by beginning programmers. The "D" stands for *dummy,* as in dummy section.

REL is the pseudo-op used to generate relocatable code using Apple's special relocating loader. Most of today's code for the Apple II will only run in one place and does not need the extra complications of relocatability. More details in the EDASM manual.

EXTRN and *ENTRY* give you ways to either use an outside label or pass a label to the outside. These see little use in shorter programs. They do get most handy if you are splitting things up into modules that have to work with each other or with a supervisory program.

Once again, see Appendix A for structure pseudo-ops unique to "new" EDASM.

File Pseudo-Ops
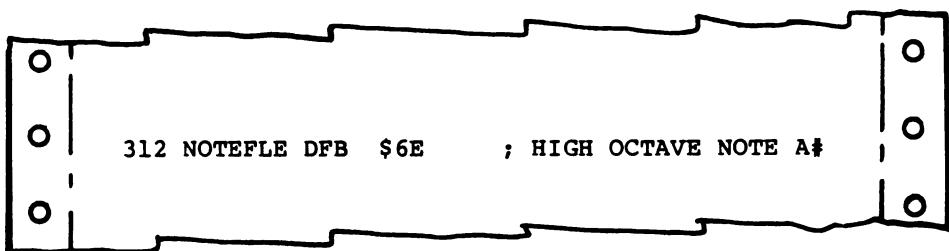
The file pseudo-ops are used to help you build files, and other parts of a program that do not use op codes. There are two important ones, and four that see lesser use.

Files are the main way you have of providing lots of data inside a machine language program. You might have ASCII text files, tables of addresses, shape or sprite information, musical notes, table lookup values, or most anything else in your working files. The best and most general programs will use fairly short machine language program code to control large and easily changed files.

DFB
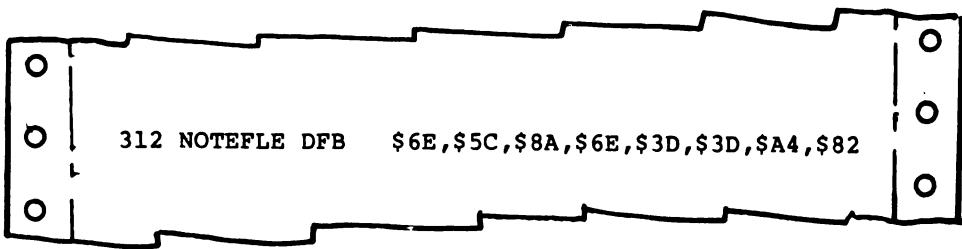
The pseudo-op DFB stands for *define byte.* This gives you a way to enter one or more file values.

For instance, the source code line . . .

```
312 NOTEFLE DFB  $6E      ; HIGH OCTAVE NOTE A#
```

. . . puts the value hexadecimal $6E into the file whose starting address equals NOTEFLE. The operand here can be in hex, in decimal, or can even be a label. Whatever you use, the operand must be convertible to an 8-bit data value ranging from hex $00 to $FF, or from decimal 0 to 255. Sometimes you will put a label on each file entry, while other times you just name the base address at the start of the file.

Should you want to put bunches of note values into your note file, use commas and keep going . . .

```
312 NOTEFLE DFB    $6E,$5C,$8A,$6E,$3D,$3D,$A4,$82
```

. . . In this case, we end up with eight file values stuffed into the file called NOTEFLE. Value $03 goes into the first location, $05 into the second, and so on. In this case, the label only applies to the *first* data entry. You can hit the $05 with a "calculated" NOTEFILE+1 label and the $07 with NOTEFILE+2 label, and so on.

For best appearances, limit your DFB file entries to eight per line. Use no spaces between entries, and don't forget that "$" sign for hex values. Normally, you do not use comments when you have long file entries, since the printout will get messy. It is permissible to "pad" your first data value with three spaces after the DFB. This moves all the data values over into the comment field for a cleaner listing. Any extra spaces between DFB and the data values will get ignored by EDASM.

Be sure to *not* end your data values with a trailing comma. This confuses EDASM and might throw in an extra $00 value on the end. And don't forget those "$" signs. It's real easy to leave one off, plowing up a single data value. Also, be sure to *not* have spaces between your file entries.

If you are involved with very large or very long files, it might be better to skip over the file area during the assembly process, and then load the file as a block off your disk. A file of a hundred or more characters takes a lot of typing and burns up a lot of disk space.

Sometimes other programs, such as a word processor, a data base manager, or a custom file generator, might be a better way to build very long or very involved files.

For instance, say you needed a cosine table for a plotter's "circle fill" routines. You could use Applesloth to calculate the values and then insert them as needed, along with commas and dollar signs, into a text file that is readable by your assembler, or else in a ready-to-use binary file format. The final code is then inserted into your machine language object program as needed. Long files most often go at the end of a machine language object program and are thus easily appended.

Note that DFB simply puts *data* values into program locations. Most machine language programs have two areas of code. One area contains op codes pieced together into legal 6502 commands. The other area holds numbers, characters, or other data values that are accessed and used by the working code portion of the program.

It is very easy to get EQU and DFB mixed up. Here's the rule . . .

> **Use EQU to equate values and address labels BEFORE your "real" op codes.**
>
> **Use DFB to define files and file entries DURING or AFTER your "real" machine language op codes.**
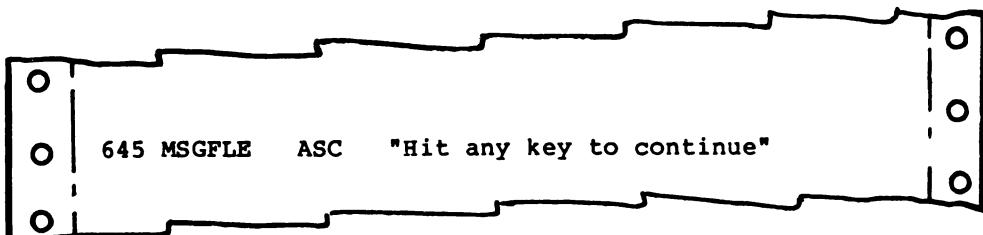
Normally, you *pre-establish* values and op codes ahead of time with the EQU pseudo-op. You *post-define* files and file entries with the DFB pseudo-op.

Should you have a short stash inside your running machine language object code, you put your DFBs *inside* the code as the need arises. Just be certain that you use these stashes as stashes and don't try to execute them as op codes.

ASC

This pseudo-op is short for ASCII, and lets you create a file of ASCII characters. The first and last characters of the operand are *delimiters* that separate the message from any comments later in the line. The delimiter symbol must not be included in the ASCII string.

For instance, the source code line . . .

```
645 MSGFLE    ASC    "Hit any key to continue"
```

. . . will put the code for an ASCII capital "H" in the first location of the message file, the code for a lowercase "i" in the second location, and so on. In this example, the quotes are the delimiters. Everything *after* the first delimiter and *before* the second delimiter is put into memory as ASCII characters. Should you want to print a quote, you use a different delimiter symbol pair, such as a "/" (slash) at the start and end of your text string.

The delimiter is not allowed to appear in the text message. It is usually a bad idea to tack a comment on the end of an ASC pseudo-op since the printout will get messy. An interesting quirk of EDASM is that you do *not* need the second delimiter if you have no comments. But watch out for tail-end spaces if you try this.

Once again, if you have very long text messages that aren't going to change much, you are better off generating those messages with a word processor and then separately combining the message files into the assembled program. Otherwise you are into lots of typing and need mucho disk space for your source code programs. Another advantage of "new-way" word processing is that you can easily include uppercase and lowercase in your messages, even with older Apples.

Provided, of course, that everyone who is going to use your program has some way to display lowercase.

ALSO RANS —

This time, our less-often used pseudo-ops give us special ways of entering special things into files.

MSB lets you clear or set the most significant bit of any ASCII characters that are generated by the assembler. MSB ON sets that bit, and MSB OFF clears that bit. For most Apple uses, ASCII has its MSB set. ASCII with its MSB off is sometimes called *low ASCII,* while a set MSB is called *high ASCII.*

The *DCI* pseudo-op lets you build ASCII files that use low ASCII for all but the *last* character and high ASCII for the final character. Which gives you a very special-use way of telling when you get to the end of a message. This slightly shortens text messages since you may no longer need a special "end-of-text" marker, such as an ETX or a NUL.

The *DW* pseudo-op defines a double-byte, or a 16-bit word put in memory backward, the way a 6502 expects a low, then a high, address. This is most used for lists of addresses.

The *DDB* pseudo-op lets you define a double-byte, or 16-bit word put in memory exactly the way you show it in the operand.

Use DDB for "frontward" entry. Use DW for "backward" entry for 6502 addresses.

The *DS* pseudo-op stands for Define Storage. This one gets used to set aside an area for file values without actually defining those values at the present time.

Conditional Pseudo-Ops

There are three special pseudo-ops in "old" EDASM that let you do *conditional assembly.* Conditional assembly means that you can assemble only certain parts of your programs under certain conditions.

Conditional assembly is very specialized, but is most useful when you want to use one master program to handle several different options of final object code. Since this is an advanced technique, we aren't going into too much detail here.

The three pseudo-ops involved are called DO, ELSE, and FIN, and are always used in their alphabetical order. See the EDASM manual for more details.

On "new" EDASM, there are many new and more powerful conditional execution commands. See Appendix A.

YOUR OWN ASSEMBLER

That just about sums up the pseudo-ops. We have been careful to show you the more important ones, and those that you are more likely to want to use early in the game. We have seen that a pseudo-op is a special instruction that we send to the assembler for use during assembly time.

We can arrange pseudo-ops into pretty printing, structure, file, and conditional assembly groupings.

The pretty printing pseudo-ops are used to make the final printout attractive, and include PAGE, SKP, LST, and SBTL.

The structure pseudo-ops are used to organize the source file assembly process, and include ORG, EQU, and CHN.

The data pseudo-ops are used to create data files of numbers or text characters. Of these DFB and ASC are the most important.

Finally, we have conditional pseudo-ops that let you assemble only parts of a program. These are not needed for most simple assembly jobs.

I've tried to present things in a totally different way than the EDASM manual does. If you have any problems, be sure to check into their way of presenting things as well.

If the right one don't get you then the left one will.

But what if you are going to use a different assembler? You'll find things pretty much "alike but different somehow."

Go back through this chapter, and any place you find an identical pseudo-op or way of doing things, paint it green with a transparent page highlighter. If there are obvious differences, paint the problem areas pink with another highlighter. If the difference is minor, note it in the margin. Then list the important differences in the box on the next page.

Things may look a little fuzzy at this stage of the game.

Blurry even.

Our ripoff module examples should clear things up. Before we find out how to write a source code file, though, we need ways to organize the source code lines into something that is readable and works. We can call this our . . .

DIFFERENCES BETWEEN MY ASSEMBLER AND EDASM:

3

SOURCE CODE STRUCTURE

We now know that the source code is a script, or a series of instructions that we send to an assembler program. The assembler, in turn, takes these instructions and uses them to generate object code, or a runnable machine language program. We also know that the source code is made up of code lines, and that each line has a number field, a label field, an op code field, an operand field, and a comment field. Very special rules have to be followed for the use of each field.

What next?

Obviously, we have to decide which lines go where in your source code. But, where do you begin? Exactly how do you arrange your source code so that it will both generate useful object code for you and still keep it understandable and self-documenting?

Think about the *structure* of your source code . . .

STRUCTURE—

The sequence and arrangement of lines in a program or source code.

Also, any methodology for writing programs or source code.

Structure is nothing more nor less than how you arrange a particular program. In the case of assembler source code, structure involves which lines go where in what order.

Now, structure may sound great, but it is really one of the most dangerous and most insidious things you would *ever* want to let appear in *any* computer program. One of the obvious reasons is that . . .

> **ANY ATTEMPT WHATSOEVER IN USING STRUCTURE IN A PROGRAM WILL SURELY LIMIT YOU ONE WAY OR ANOTHER AND IS CERTAIN TO RETURN TO HAUNT YOU!**

Structure may limit how fast you can do things. Or, it may force your program to take up too much room. Or, it may restrict you from doing what you want to do in the way you want to do it. Structure may simply ignore most of the resources available to you.

Remember, the whole truth and beauty of machine language code is that anything goes! And, if structure ever gets between you and anything, that's ungood. In fact, if you are to become a decent machine language programmer, the foremost tenet of your beliefs absolutely *must* be that . . .

> **ALL STRUCTURE IS INHERENTLY AND INTRINSICALLY EVIL!**

Structure, of course, is what makes BASIC so bad, Pascal utterly ludicrous, and what renders Ada totally unspeakable in polite company.

Why, using structure is almost as bad as using a road.

If you are traveling on land, you will almost always have a more challenging trip, a more informative trip, and a vastly more creative trip if you do *not* travel by road.

First, the road will take you where it wants you to go, rather than where you really should be headed. Secondly, the road will very severely limit your choice of a mode of transport. Using a road also drastically restricts your possible travel speed to a very narrow range. Contemplate a sunset in the fast lane sometime and see what happens to you.

While in a canoe.

A road will also have all sorts of silly regulations as to which side to travel on, access rules, license demands, speed limits, vehicle restrictions, and lots of other dumb things that go with the totally nonsensical concept of structure. Needless to say, you *always* end up paying to use a road, one way or another.

But the most insidious and most despicable thing about a road is that others will have traveled it before you, still others will travel with you, and yet others will travel it after you. Which very severely demeans the experience for all.

So, structure is inherently and intrinsically evil. Particularly if it gets between where you are and where you want to go.

Or become.

As a machine language programmer or an assembly language programmer, you are totally free to pick and choose as much or as little

structure to your programs as you feel comfortable with. But, don't ever go along for the ride just because there's some stupid and misguided attempt at structure that "they" want you to use.

What we are really saying is . . .

> **Go ahead and use any structure you may feel comfortable with,**
>
> **BUT—**
>
> **Be ready to fire bomb the mutha at any time for any reason.**

So, now that I've shown you how bad structure is, I'll show you some structure.

It is the structure I use for my assembly language programs. Just as with any structure, mine is severely limiting and restrictive. The only thing nice you can say about it is that my structure is orders of magnitude less limiting than BASIC, and infinitely less restrictive than Pascal.

So, use my structure if you like. If not, go ahead and do anything you like.
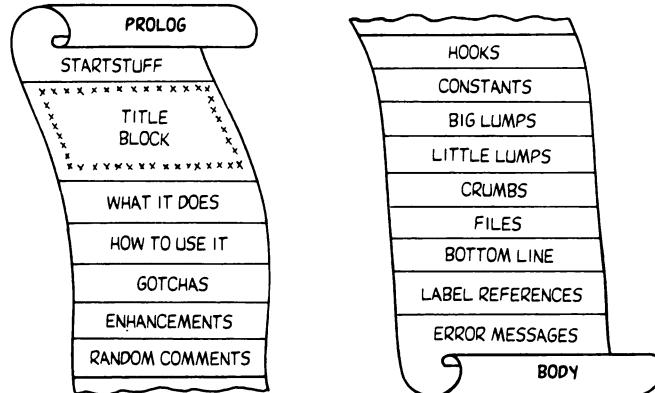
Remember . . .

> **Anything goes in machine language!**

Anyway, the goals of the structure I use are, first, to make the assembler put together the object code in a reasonable way, and secondly, to be attractive and understandable to people for later use.

There are 16 parts to a source code listing that follow my structure . . .

SOURCE CODE STRUCTURE:

To make this structure easy to understand and use, it has been put into a ripoff module called the EMPTY SHELL.SOURCE, and will be our first ripoff module in Part II.

Many of today's assembler listings seem all cramped together and hard to read. The main reason for this is that the programmers are attempting to save on source code length or are otherwise suffering from some form of dino dementia. With most any modern Apple

assembler, there is no reasonable limit to the length of source code. So, any time some blank space or some formatting commands make the source code read better or seem better thought out or better organized, use them . . .

> **NEVER try to save source code file space at the expense of legibility or clarity.**
>
> **ALWAYS spend the time and space you need to create a source code file that is easy to understand and pleasant to read.**

Another essential rule is that . . .

> **More than one half of your source code should be documentation aimed at informing the reader or user.**

Remember that source code must serve two ways. First it has to be usable by an assembler to generate object code. But secondly, and equally important, source code must be legible and understandable by a user or customer. The source code should always explain exactly what a program does, exactly how a program does it, and exactly how a program can be modified, either for an upgrade or for a move to a different machine.

By the way, you will just about *never* make a hard copy of your source code. Instead, you use a combined source and object code printout that gets generated "free" when you do an assembly. This combined source and object code is called an *assembler listing* . . .

> **ASSEMBLER LISTING—**
>
> **A hard-copy printout that combines source and object code onto one single document.**
>
> **Assembler listings are the most common way of showing source code.**

The assembler listing is by far the most common way of keeping hard-copy records of the assembly process. Any time you read a magazine article that includes machine language code, you'll find an assembler listing of one kind or another present. We'll see more details on reading an assembler listing later.

One big advantage of assembly listings over "pure" source code is that assembler listings show you the generated machine language object code that goes directly with the source code. You can often spot nonsense op codes this way, and the versions won't get mixed up as well. You also cannot normally get an error-free assembler listing of something that will not assemble. This forces you to save only records of code that will at least assemble properly.

The EDASM assembler automatically generates assembler listings

during assembly if you turn the printer on with a "PR#1" command. Other assemblers will do the same thing one way or another.

You rarely would want to make a printed copy of your source code for a record or for publication. You create source code on screen and save source code to disk or to RAM for assembly, but you use assembler listings for your records and final hard copy . . .

> **Hard-copy printouts of "pure" source code are rarely used or needed.**
>
> **Instead, use assembler listings for your records and publication.**

There is one little hangup about assembler listings, though. Remember those pseudo-ops? An assembler listing will show you the *result* of a formatting pseudo-op, rather than listing it . . .

> **Certain pseudo-ops, such as PAGE, SKP, and SBTL will not appear on an assembler listing. Neither will their line numbers.**
>
> **Instead, you will see the results of these pseudo-ops instead, such as a new page, numbered headers, or skipped lines.**

So, if you have a line number missing on an assembler listing, chances are that line number was a pseudo-op intended to format things one way or another. If line 25 is missing and replaced with six white spaces, obviously a SKP 6 is inferred here where the "missing" line belongs.

An important point . . .

> **Your assembly listings should be works of art that must stand on their own merits, separate from any code they may generate.**
>
> **You should be as concerned about layout, appearance, and how images fit the page as would a calligrapher or poet.**

There are two halves to my source code structure. The *prolog* mostly holds documentation intended for the user, while the *body* mostly holds commands for the assembler . . .

> **PROLOG—**
>
> **The first part of a source code listing, used mostly to inform the programmer or user.**
>
> **BODY—**
>
> **The second part of a source code listing, used mostly to hold assembler commands.**
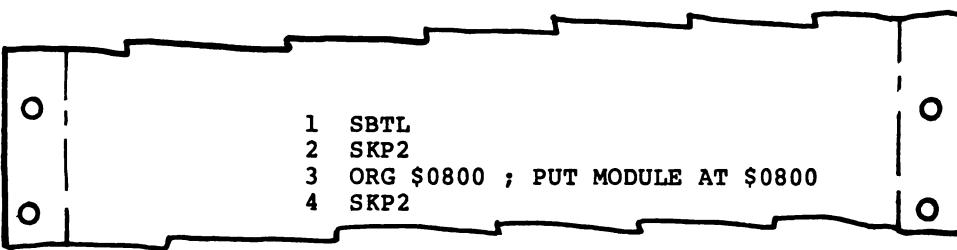
You will find a few pseudo-ops in the prolog, and there's bound to be lots of documentation in the body, but the prolog is basically for the user, and the body is basically intended for the assembler program.

Let's look at the 16 parts to my source code structure. We'll use various examples from different application programs to show what each part is supposed to do.

STARTSTUFF

The *startstuff* is a few lines of code always used at the beginning of your program to get everything off on the right foot.

Let's look at some typical startstuff source code. Here is what you might punch into EDASM . . .

```
1  SBTL
2  SKP2
3  ORG $0800 ; PUT MODULE AT $0800
4  SKP2
```

You punch in your commands, using "space" to mean "tab" on "old" EDASM, or real IIe tabs on "new" EDASM. The line numbers are provided free for you. So, when you see that "1," you hit a space or tab to get past the label field, because you do not want a label on line 1. The SBTL is the pseudo-op here, and tells the assembler to name and number each assembler listing page. The first SKP 2 tells us to put some white space between the header and the first printed line. SBTL, of course, will not print, and we will see its *result* instead.
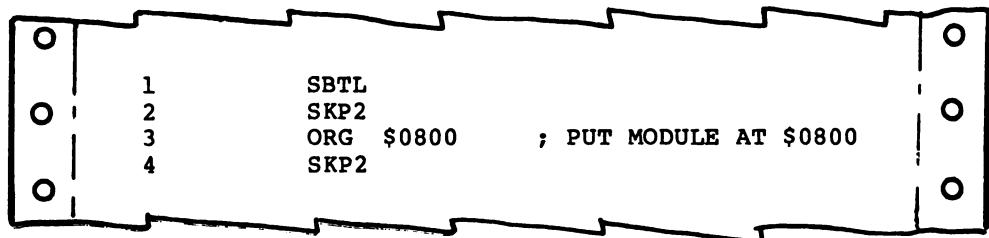
On line 3, you tell the assembler that you want this code to be assembled onto disk such that the object code will eventually load into and run starting at location $0800. The space and the semicolon are needed to ensure you are really in the comment field. The space after the semicolon is actually a space, but it is an unneeded one I throw in to make things prettier. The comment field here is intended to inform the reader or user where the code is going to run.

I prefer to put the ORG for origin command on line 3. This way, you always know where it is and are always forced to think about where you want your code to go before you write it.

The SKP 2 of line 4 is used to get some white space after the startstuff and the upcoming title block.

So, that is all there is to punching code into your assembler. You must be very careful to use spaces or tabs for tabs and must never have any spaces inside a label, op code, or operand column.
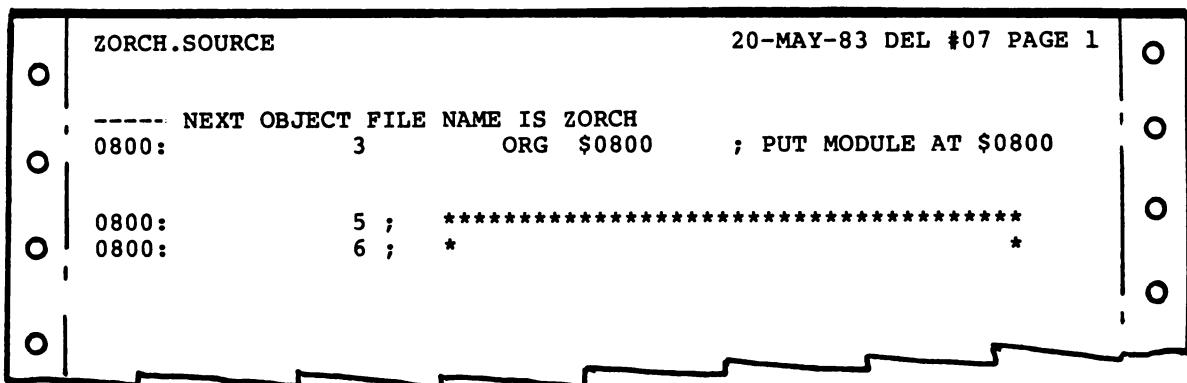
But, if you list or examine your EDASM code, you'll get everything tabbed over to where it belongs, more or less . . .

```
1           SBTL
2           SKP2
3           ORG  $0800      ; PUT MODULE AT $0800
4           SKP2
```

There are no labels in our startstuff so the label field stays blank.

When you get around to actually assembling the program, though, none of this will appear in your object code. Why? Because there are no "real" op codes so far. Just pseudo-ops. All our startstuff has done is tell the assembler to give us headers and page numbers on the listing, to start assembling into a certain location, and to leave some white space in two places.

Your assembler listing will look something like this . . .

```
ZORCH.SOURCE                              20-MAY-83 DEL #07 PAGE 1

----- NEXT OBJECT FILE NAME IS ZORCH
0800:           3            ORG  $0800      ; PUT MODULE AT $0800

0800:           5 ;  ****************************************
0800:           6 ;  *                                      *
```

And this one looks more differenter still. What happened to lines 1, 2, and 4? Why all those 0800's?

Well, we've taken a peek ahead here to show you how source code looks when first entered, when listed on-screen as source code, and when printed as an assembler listing.

Instead of line 1, we get the header and the page number. Instead of lines 2 and 4, we get the results of those lines with two skipped lines worth of white space. And, since we have yet to feed a "real" op code to our assembler, it stays "stuck" at $0800 and has nothing to put in all that white space reserved for object code bytes. Lines 5 and 6 we borrowed from the upcoming title block, just to show you what it would look like. Should you end your assembly at this point, you will get an object code that is zero bytes long.

As we'll find out later, up to three object code bytes can follow the current assembly address, and a label can fit between the line number and the operand.

Anyway, back to our structure and the startstuff.

With EDASM, I like to start with the SBTLE command, because this gives you a nice header, date, and page numbering on each page.

There is one bug with SBTL, though . . .

> **The EDASM pseudo-op "SBTL" has a bug that causes certain printers to tab off-page if "old" EDASM itself produces a formfeed.**
>
> **One way to beat this is to always force your own next page with a PAGE pseudo-op.**

Which simply says to call your own page breaks on a "new" or "old" EDASM assembly listing. Calling your own page breaks is a good idea anyhow for best legibility. Use the PAGE pseudo-op to do this.

Anyway, so much for SBTL. You own assembler may need different setup commands. If so, change the startstuff to suit your needs.

The next part of the startstuff is the origin line.

> **ORIGIN—**
>
> **That place in memory where the object code is to actually begin.**

Things get sticky fast here. On a disk-based assembler, you can give an origin command that goes anywhere in memory, even overwriting the assembler code. You can do this since the code gets put onto disk, rather than assembled directly into your machine. To get the object code into your machine, you load or boot the disk in the usual way.

Some in-place assemblers give you the option of assembling your code in one part of the machine with the intention that it eventually is to run elsewhere. This keeps you from overwriting the assembler during the assembly process. A block move of some sort is needed to relocate the object code in its final running position. These in-place assemblers will ask you for a *destination* address along with a final *object code* address instead. "New" EDASM offers this as an option.

Unless you go to a lot of trouble to make your object code relocatable, it will run only at one location in the machine. In fact . . .

> **EDASM demands an origin for the object code, or it will not give you any assembly at all.**

Many dino assemblers will bury the ORG command in the middle of the source code somewhere. I like to always put it on line 3. That way, you always know where it is, which makes for easy finding and easier changes. You also get the "- - - - - NEXT OBJECT FILE NAME IS ZORCH" line which keys the name of your source code to the name of your object code.

As a reminder, object and source codes must have different names, since one is a series of script-like instructions and the other is ready-to-run code. If you do nothing to EDASM, a source code called ZORCH will assemble into an object code called ZORCH.OBJ0. I

much prefer to tell the assembler to assemble ZORCH.SOURCE into ZORCH. More on this when we get to actually assembling programs.

You can also assemble your source code in several pieces. While you normally should try to put the whole object code together into one single piece, split object codes can have special uses on larger and longer programs.

If you are using more than one piece for your object code, a new ORG statement is needed for each piece.

EDASM does this one of two ways. If you *absolutely* define a new origin, you get a new object code file, usually named ZORCH.OBJ1, then ZORCH.OBJ2, and so on. An example of an absolutely defined origin is ORG $8A00.

On the other hand, if you *relatively* define a new origin, EDASM generates only a single object file, filling in with garbage between the old and new object code. One way to define a new relative origin is to put some label, say START on your first line that holds a "real" op code. Then a source code command of ORG START+$0300 will automatically add garbage at the end of the original object code, and start the new stuff off exactly at the origin plus $0300.

You could also back up, but this would overwrite generated object code and would normally be kind of dumb.

At any rate . . .

> **EDASM lets you generate more than one object code from a single source code, or else lets you move around and leave "holes" in one long object code file.**
>
> **A *relative* new origin, such as ORG START+$0300 generates one long object code file with holes in it.**
>
> **An *absolute* new origin, such as ORG $7600 generates a second and separate object code file.**

Normally, you use a single object code without any holes in it. Multiple or continuous object code should be reserved for special uses only.
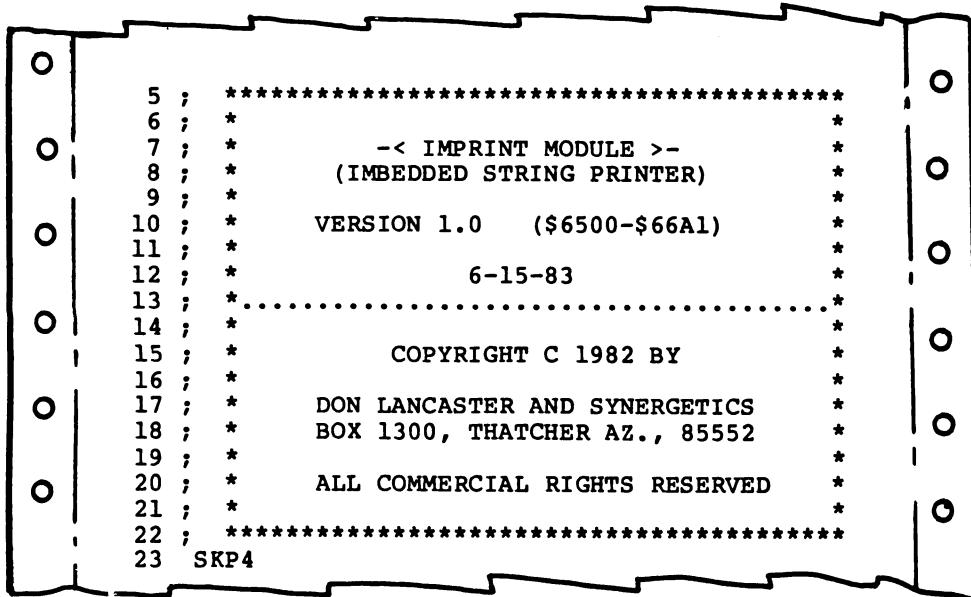
At any rate, your first origin pseudo-op in your program must be an absolute one, and must be done before any source code lines involving real op codes appear in your source code. This first ORG must exist, or EDASM won't generate any object code at all.

Once again, the purpose of the startstuff is to decide where the object code is to be assembled, to set up headers and page numbers, to give us some pretty white space, and to do anything else you need to get your assembler started off in the right direction.

TITLE BLOCK

The next "piece" to the structure of an assembly program *is the title* block. The title block should name the program, give a hint of what it does, show the date and version number, and should include a copyright notice.

Here's an example of a title block . . .

```
 5 ;  ***********************************************
 6 ;  *                                             *
 7 ;  *            -< IMPRINT MODULE >-             *
 8 ;  *            (IMBEDDED STRING PRINTER)        *
 9 ;  *                                             *
10 ;  *        VERSION 1.0    ($6500-$66A1)         *
11 ;  *                                             *
12 ;  *                 6-15-83                     *
13 ;  *.............................................*
14 ;  *                                             *
15 ;  *            COPYRIGHT C 1982 BY              *
16 ;  *                                             *
17 ;  *        DON LANCASTER AND SYNERGETICS        *
18 ;  *        BOX 1300, THATCHER AZ., 85552        *
19 ;  *                                             *
20 ;  *        ALL COMMERCIAL RIGHTS RESERVED       *
21 ;  *                                             *
22 ;  ***********************************************
23  SKP4
```

The fancy box makes a good and attention-getting way of setting the title essentials away from the rest of the source code. Since most programs have funky or catchy names, you might qualify the name with a second descriptive line as shown here.

You also should provide a version number, a version date, and the place in memory where the object code is going to run. Don't forget that object code can go a few bytes beyond the last assembled line number if that happens to be a 2- or a 3-byte instruction or if there is a stash or a file at the end. If you don't know the length of your program ahead of time, take a guess, and correct it later.

The bottom half of your title block should have a copyright notice, your name and address, and a use disclaimer.

By the way, all software printed listings are protected automatically and free by an upgrade of what once was "common law" copyright until such time as they are sold. Theoretically, there are legal remedies to people ripping off your work. But . . .

> **In the real world, you as software author, have ZERO recourse to any ripoff, use or misuse of any code you may create.**

Any large company, the Feds, or for that matter, anyone else with a lawyer and more money than brains, can immediately and totally deprive you of any rights you may have naively thought you might have had regarding software authorship. At the very least, the time value of any possible recourse can be made so painfully drawn out and so ridiculously expensive as to be less than worthless.

This is reality. All else is bull- - - -.

What you do as a software author is hope for the best. Should you get ripped off, pick yourself up, and start over again, putting your energy and personal value added into new work rather than into an

interminable psychic energy sink. Recognize that *anything* "legal" can be weaseled out of one way or another, or else can be delayed long enough to become moot.

But the copyright notice does one very important thing. It makes it quite clear to anyone who is stealing your work that they are in fact stealing, rather than reusing public domain material or adapting material freely given to them.

So, be sure to put that copyright notice on your listing. At the very least, it might keep you from being attacked by a leopard or hit by a meteor.

Maybe even both.
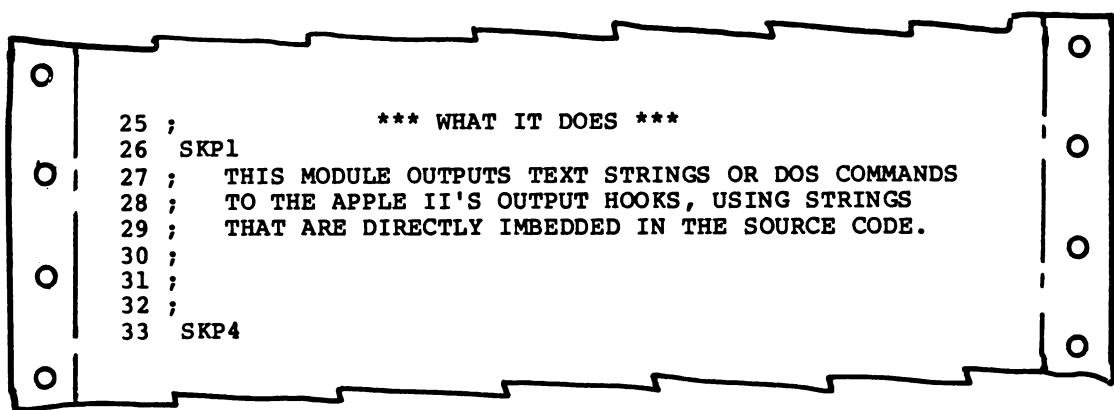
Usually, it is far easier to edit an existing source code than it is to create a new one. So, typically you will steal a title block off something you already have done and edit it into something new.

The title box, of course, is all comment. Remember that you start an all comment line with either a ";" or an "*". I prefer the ";" since it isn't as obnoxious and is somehow less dino looking.

WHAT DOES IT DO?

Next, add as many comment lines as you need to tell us exactly what the program does. Keep this description simple, explicit, and mainstream.

Here's an example . . .

```
25 ;              *** WHAT IT DOES ***
26 SKP1
27 ;   THIS MODULE OUTPUTS TEXT STRINGS OR DOS COMMANDS
28 ;   TO THE APPLE II'S OUTPUT HOOKS, USING STRINGS
29 ;   THAT ARE DIRECTLY IMBEDDED IN THE SOURCE CODE.
30 ;
31 ;
32 ;
33 SKP4
```
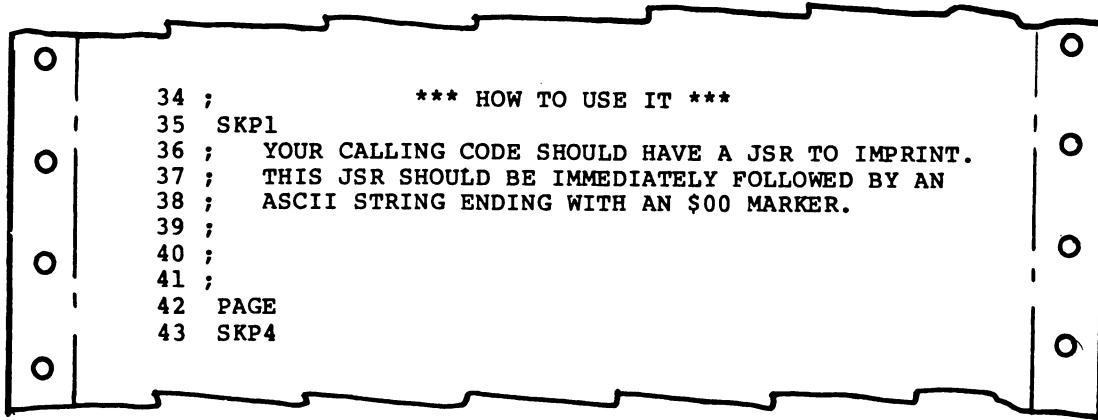
This description should apply to the main aim of the entire source code. Later on you will have lots of chances to show what the separate parts of the code do individually. I like to provide for six lines here. That's what the final three "empty" comment lines are for. The first SKP gives you some white space between title and description. The second separates this part from the next part of our structure, which tells us . . .

HOW TO USE IT

Next, tell us exactly how to use the object code. Tell us everything you need to know to use the main part of the code in the usual way you want it used. Again, hold off on specific details on special use modes. Tell us only the mainstream stuff as compactly as possible.

Something like this . . .

```
34 ;              *** HOW TO USE IT ***
35  SKP1
36 ;    YOUR CALLING CODE SHOULD HAVE A JSR TO IMPRINT.
37 ;    THIS JSR SHOULD BE IMMEDIATELY FOLLOWED BY AN
38 ;    ASCII STRING ENDING WITH AN $00 MARKER.
39 ;
40 ;
41 ;
42  PAGE
43  SKP4
```

As with the title block and the "what-it-does" module of our structure, this block is pure comment, intended for people and ignored by the assembler program. Tell us here *only* the essentials of how to use the code that follows.
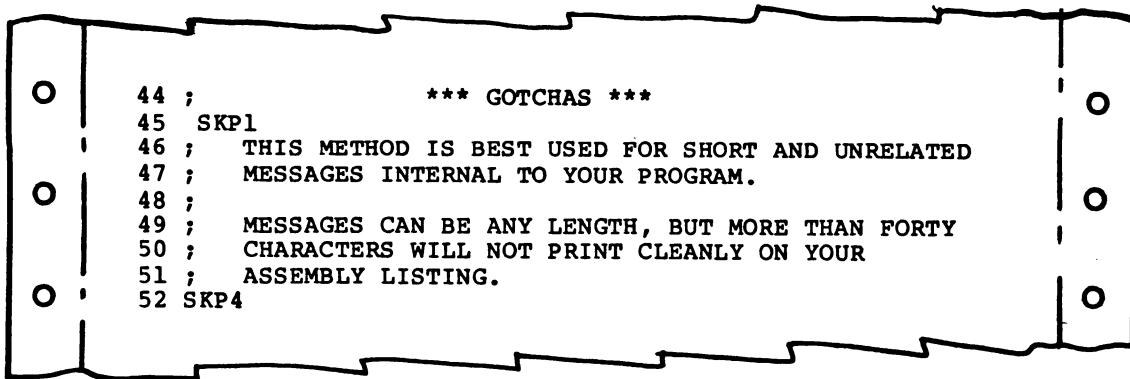
The PAGE command is used on line 42 because you most likely have put all you want to on the first page of your assembler listing. By the way, the notation we have shown you previously is the actual source code as you place it on screen. The assembler listing will have white spaces for the SKPs, page breaks for PAGE, and generated object code for "real" op codes.

There are three more "pure comment" modules . . .

GOTCHAS

A *gotcha* is anything that will make the program hang up or not work. Tell us here how much RAM you need in the Apple. Can a language card be used? Must a language card be used? Do you need Applesoft ROMs? Will it only run on a IIe? Does Eurapple timing make a difference? Is lower case necessary? Is a Flugelhoph card in slot 5 allowable? Are other code modules needed to let this program work? Will the program bomb if the paddles aren't plugged in? Are hardware mods required?

Here's an example of some gotchas . . .

```
44 ;              *** GOTCHAS ***
45  SKP1
46 ;    THIS METHOD IS BEST USED FOR SHORT AND UNRELATED
47 ;    MESSAGES INTERNAL TO YOUR PROGRAM.
48 ;
49 ;    MESSAGES CAN BE ANY LENGTH, BUT MORE THAN FORTY
50 ;    CHARACTERS WILL NOT PRINT CLEANLY ON YOUR
51 ;    ASSEMBLY LISTING.
52 SKP4
```

It pays to keep the fancy title and stars on each and every one of the
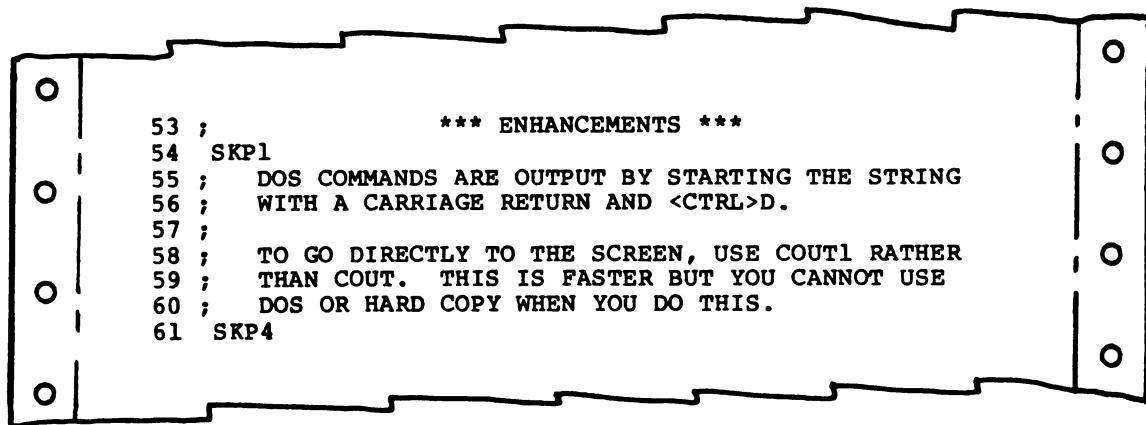
comment modules. This adds an overall consistency and style to all your programs, and forces you to put what is essential into the proper place.

Note also how a comment that is too long for one line can be broken up and done in several sequential lines. But remember that this is a *line-oriented* assembler. There is no way that anything "left over" at the end of one line will automatically get picked up and used on the next line down.

ENHANCEMENTS

An enhancement is the opposite of a gotcha. What new can you do with the program that is above and beyond what you intended? How are these features activated? Why would you want to use them?

Again, here's an example . . .

```
53 ;                    *** ENHANCEMENTS ***
54 SKP1
55 ;    DOS COMMANDS ARE OUTPUT BY STARTING THE STRING
56 ;    WITH A CARRIAGE RETURN AND <CTRL>D.
57 ;
58 ;    TO GO DIRECTLY TO THE SCREEN, USE COUT1 RATHER
59 ;    THAN COUT.  THIS IS FASTER BUT YOU CANNOT USE
60 ;    DOS OR HARD COPY WHEN YOU DO THIS.
61 SKP4
```
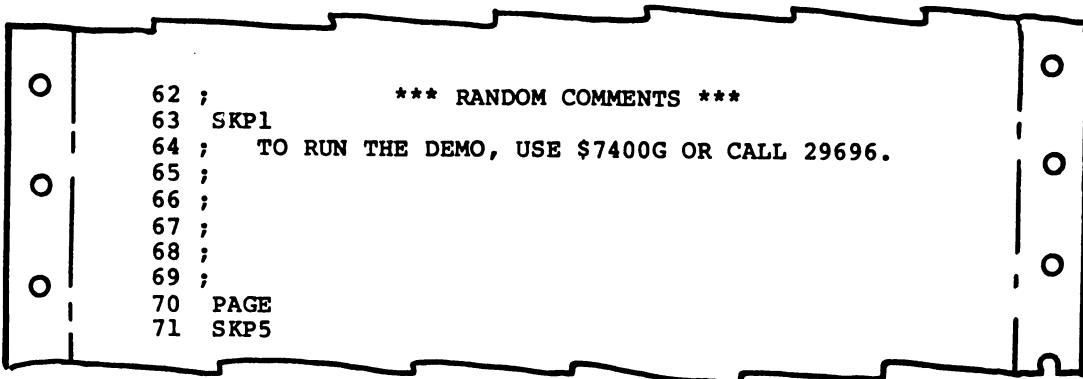
Try to keep things that just about any user needs in the "what-it-does" and "how-to-use-it" modules. Save this module for details for the dedicated user who wants to go beyond what is obviously offered. Again, there will be room below for comments specific to smaller portions of the whole program.

Always keep your comments simple, straightforward, and to the point. Provide only what is needed, but *everything* that is needed. Do so as concisely as possible.

And, if you need to say more, add some . . .

RANDOM COMMENTS

If there is some other stuff you want to say about your whole program, here is where you cram it in. Again, we have a module that is pure comment . . .

```
62 ;              *** RANDOM COMMENTS ***
63 SKP1
64 ;    TO RUN THE DEMO, USE $7400G OR CALL 29696.
65 ;
66 ;
67 ;
68 ;
69 ;
70 PAGE
71 SKP5
```

We'll note in passing that this particular demo is just a demo, and something that will not be used after you have studied it. Thus, this goes as a random comment, rather than in the "how-to-use-it" block.

So far, just about every line has been spent telling the user about something. Only our origin in the startstuff is something needed by the assembler program. Everything else is comments by one person for other people.

And that pretty much wraps up the prolog half of our source code structure. The prolog gets things started, tells us what the program is, and then tells us how to use it. The body is more concerned with getting input that the assembler can use, although it will also have lots of comments present.

Let's now pick up our first body module that both the assembler program and people can use . . .

HOOKS

The hooks are "connections" that you want to make to known locations in the Apple II . . .

> **HOOKS—**
>
> **Labeled connections made to known and fixed Apple memory locations.**

As review, one of the big advantages of a symbolic assembler is that you can put labels on things. These labels can refer to values, to locations in a program, or to specific fixed locations in the Apple II or IIe. A hook is how you pre-define a specific location for use by the program.

A hook source code line will start with the defining label, followed by the pseudo-op EQU, followed by the absolute address. Later on, operands in the program can refer to these hooks. It is best to define the hooks *before* you use them although a good assembler will find a label anywhere in the source code.

One exception in "new" EDASM: If you specifically want to force an absolute address that references page zero, your EQU must *follow* the first operand label reference. Otherwise, you will get page zero addressing, rather than absolute addressing. The need for an absolute address from $0000 to $00FF is very rare, but can solve some timing problems.

Here are some hooks . . .

```
72 ;                    *** HOOKS ***.
73  SKP1
74 COUT    EQU   $FDED      ; OUTPUT CHARACTER VIA HOOKS
75 HOME    EQU   $FC58      ; CLEAR SCREEN
76 KBDSTR  EQU   $C010      ; KEYBOARD RESET
77 INIT    EQU   $FB2F      ; INITIALIZE TEXT SCREEN
78 KEYIN   EQU   $FD1B      ; READ KEYBOARD
79 PRBL2   EQU   $F94A      ; PRINT X BLANKS
80 SETINV  EQU   $FE80      ; SET INVERSE SCREEN
81 SETNORM EQU   $FE84      ; SET NORMAL SCREEN
82 STRP2   EQU   $EB        ; POINTER TO ASCII STRING
83 WAIT    EQU   $FCA8      ; TIME DELAY SET BY ACCUMULATOR
84          SKP3
```

For instance, we see that the label WAIT is defined as absolute address $FCA8, which is the Apple II location for a monitor delay subroutine.

Hooks may be defined as an 8-bit value or a 16-bit value. An 8-bit value is interpreted by EDASM as a page zero address. A value of four hex bytes, or 16 bits is interpreted by EDASM as an absolute address. The operand can modify the addressing mode by adding a comma for indexed addressing, or including parentheses for indirect addressing modes, and so on.

Hooks may also be used to set aside locations on page zero for special use by your program, and can also be the interconnections between your program and other modules, possibly even in other languages. In line 82, STRP2 is a string pointer we have defined and set aside on page zero for use by this program module. Note that the same label can point to the *next* location as well, by using STRP2+1 and doing operand arithmetic.

Three gotchas. Be sure to use the "$" sign for hex addresses. Do not use a "#" symbol when defining a hook. The "EQU" says that you are defining a value. Finally, all EDASM page zero locations *must* be pre-defined with hooks before they are used.

Summing up . . .

USING "EQU" HOOKS
DON'T forget the "$" sign in front of hex addresses.
DON'T use the "#" symbol, as EQU says nearly the same thing.
DO pre-define all EDASM page zero addresses using 8-bit hook values.

As review, do you remember the difference between an EQU and a DFB? Well . . .

> **EQU** means *equate* and pre-defines an address before it is used.
>
> **DFB** means *define* and enters an in-place value into the object code.

Thus . . .

> **EQUs** come BEFORE any real op codes.
>
> **DFBs** appear DURING or AFTER any real op codes.

One exception: On "new" EDASM, absolute 16-bit references to page zero locations must have their EQU *following* their first use. The need to force an absolute address from $0000 to $00FF is very rare, but this rule lets you do it.

What we say for DFBs goes double for DDBs and DFWs. Why?

Any time you use an EQU, you have to match the word size and the instruction mode to what you need. For instance, on an 8-bit value, such as WNDTOP EQU $20, a LDA WNDTOP will load the accumulator from page zero memory location $0020. On the other hand, a LDA #WNDTOP will put the immediate value hex $20 into the accumulator.

As 16-bit examples, say you use a DISK EQU $C08C. A command of LDA DISK will load the accumulator from absolute location $C08C. A command of LDA DISK,X will load the accumulator from the sum of what is in the X index register and absolute location $C08C. Typically, there will be a $60 in the X register for slot number six, and the LDA DISK,X will access absolute location $C0EC, which reads the diskette if things have been set up properly. This use of indexed instructions is the usual way to set up slot independent code.

The rule is this . . .

> **ALWAYS** match the size of the EQU to the size of the needed 8-bit or 16-bit value or address.

There is one sneaky way you can split up a 16-bit EQU into two 8-bit pieces. This is done with a ">" or a "<" in the operand.

For instance, say you want to take a 16-bit previously EQU'd address and put it into a pair of 8-bit locations on page zero. Here is how to do it . . .

```
LDA >ADDRESS

STA POINTER

LDA <ADDRESS

STA POINTER+1
```

What this tells you to do is take the bottom half of a 16-bit address

and store it in a page zero location called POINTER. Then take the top half of the 16-bit address and store it in the next available location, found by adding one to POINTER.

Alternately, you could store the bottom half in POINTLO and the top half in POINTHI. Remember that most addresses in most 6502 uses are *backward,* with the low 8-bit or position address first, and the high 8-bit or page address following.

Here's the rule on the "pick half" commands . . .

> A ">" in an EDASM operand says to pick and use the lower eight bits of a 16-bit value.
>
> A "<" in an EDASM operand says to pick and use the upper eight bits of a 16-bit value.

Note that the arrow or caret points to the half of the 16-bit word you want to use. Other assemblers will most likely have some similar way to split up 16-bit words into 8-bit pieces.

Your EQU hooks should normally be put in alphabetical order, unless some other logical grouping makes more sense for what you are doing.

CONSTANTS

Some programmers like to put all their EQUs into one long list. Others like to put all the 8-bit stuff in one pile, and the 16-bit values into another. I prefer to put all the EQU *addresses* into one combined hook list.

Then, on the same page, if you have any EQUs that are fixed constants or other pre-defined but non-address values, you add a second list for these.

For instance, you might like to add *textfile commands,* such as a label "B" for backspace, "F" for formfeed, "P" for a flashing screen prompt ($60), "G" for a bell or "gong," and so on.

Here's a ferinstance . . .

```
85 ;                *** TEXTFILE COMMANDS ***
86           SKP1
87 B         EQU  $88      ; BACKSPACE
88 C         EQU  $8D      ; CARRIAGE RETURN
89 D         EQU  $84      ; DOS ATTENTION
90 L         EQU  $8A      ; LINEFEED
91 P         EQU  $60      ; FLASHING PROMPT
92 X         EQU  $00      ; END OF MESSAGE
93           PAGE
```

We will see how to use these constants later in the ripoff modules that involve themselves with printing text. We'll see then that you can use a DFB C,L instead of a DFB $8D,$8A to put a carriage return and a skipped vertical space into a message. The label method is obviously

easier to type and saves looking up the same ASCII command value a dozen times.

As another use of constants, consider a musical note program. Here you can define labels of "A1," "A1#," "B1," and so on, or something similar. The assembler can then substitute pitch values directly for the labels. But, don't forget that a single letter label of "A" is a reserved no-no. So are "X" and "Y" on "new" EDASM.

At any rate, all of the EQUs should go on one page if they will fit. The address EQUs should go first and be called hooks, while the value EQUs should go second and be called constants.

The EMPTY SHELL.SOURCE ripoff module includes hundreds of the most popular EQUs already built into the source code and ready for your use without any looking up or scrounging around. What you do is decide which EQUs you *do not* want and then eliminate all but the good guys.

BIG LUMPS

At long last, in tenth place of our assembler's source code structure, we are ready to start with the "real" program. This involves the op codes that the assembler is to use to put together some object code for us.

What you enter here, of course, depends on what you want your program to do. Regardless of how big or how small the program is, it is convenient to break the code up into three groups, which we will call the *big lumps,* the *little lumps,* and the *crumbs.*

The size of each depends on the overall size of your program. If you are writing a simple support module, the big lumps will be the actual program code, the little lumps will be the support subroutines, and the crumbs will be any sub-subs or short file entries. You can have any number of lumps of any size.

On a very large program, the big lumps will be the supervisory or high-level control code, the little lumps will be the code modules that do each individual needed task, and the crumbs will be the subroutines that do all the grunge work.

Any short or compact files that are not often changed might also be included as little lumps or crumbs. We'll have room for big files later in our structure.

Normally, it is a good idea to have your program do something on its very first address. The reason for this is that if you BRUN something off a disk, the disk will try to execute the first byte in the code as an instruction. So, you usually start with the high level big-lumps code, follow this with the lower level little-lumps code, and then work your way down to the detail or crumbs code.

Any support files are best split up so that short and rarely changed files immediately follow the module that uses them, and such that major files go on the end.

Saying it again . . .

> **The first byte in a program should normally be executable code.**
>
> **The usual way to arrange a program is heavy instructions first, then medium instructions second, and details third.**
>
> **Short and rarely changed stashes normally follow the module that uses them.**
>
> **Long files normally go at the end.**

Now, of course, anything goes in machine language. So, you are free to change and rearrange things any way you like. But, recognize that this "{big lumps}—{little lumps}—{crumbs}—{files}" structure works well for most uses. Use it unless you have a good excuse not to.

Should you absolutely have to start with a file rather than code, put a 3-byte jump at the beginning so that a BRUN command can bypass the data and get to the running commands.

Here's some good practice . . .

> **A code module ideally should have only one entry point and only one exit means.**
>
> **Ideally, the entry point should be the first byte of the code module.**

There are times and places where you may want to violate this rule, but it is generally a good idea to keep things in a modular order, with entry and exit to various modules at expected and reasonable points.

Subroutines are ideal modules, because they meet this need of "single entry at the top," and "obvious exit on return." In fact, it is a good idea to use subroutines even if the code is only needed or used in one place of a program.

The organizing power of a subroutine is even greater than its code shortening abilities . . .

> **Subroutines are nearly ideal code modules and should be used even if what they do is needed only at one point in the program.**

What single subroutines do for you is keep the crumbs out of the flow of the big-lumps code. Each detail sub can be changed without having to reassemble and retest and re-debug the high level code, provided you separate everything off into bite sized chunks ahead of time.

Two exceptions. First, if speed is everything, don't use subroutine calls and returns inside that part of the code that has to run the fastest. And secondly, it may be necessary to sometimes have two or more entry points into a single subroutine, such as for a "cold" or "warm" entry, or whatever.

There is really no difference between how you put the big lumps, the little lumps, and the crumbs into your source code. What could be a big lump for one program could be nothing but a crumb in another. In general, you put down a title, some user documentation, and then the "real" op codes.

The best examples of this appear in the ripoff modules later in the book. You might want to scan these at this time, looking for examples of how to handle these three structural parts to your source code.

Here is an example of what to look for . . .

```
181 ;                    *** IMPRINT MODULE ***
182          SKP2
183 ;        THIS MODULE UNPOPS THE STACK TO FIND THE
184 ;        IMBEDDED STRING.  IT OUTPUTS ONE CHARACTER
185 ;        AT A TIME TILL AN $00 MARKER IS FOUND.  THEN
186 ;        IT JUMPS BACK TO THE CALLING PROGRAM JUST
187 ;        BEYOND THE STRING.
188 ;
189          SKP2
190 IMPRINT  STX  XSAV2     ; SAVE REGISTERS
191          STY  YSAV2     ;
192          STA  ASAV2     ;
193          SKP1
194          PLA            ; GET POINTER LOW AND SAVE
195          STA  STRP2     ;
196          PLA            ; GET POINTER HIGH AND SAVE
197          STA  STRP2+1   ;
198          SKP1
199          LDY  #$00      ; NO INDEXING
200 NXTCHR2  INC  STRP2     ; INCREMENT TO NEXT ADDRESS
201          BNE  NOC2      ; SKIP IF NO CARRY
202          INC  STRP2+1   ; INCREMENT HIGH ADDRESS
203 NOC2     LDA  (STRP2),Y ; GET CHARACTER
204          BEQ  END2      ; IF ZERO MARKER
205          JSR  HOOK2     ; FOR SPECIAL EFFECTS ONLY
206          JSR  COUT      ; PRINT CHARACTER
207          JMP  NEXCHR2   ; BRANCH ALWAYS
208          SKP1
209 END2     LDA  STRP2+1   ; RESTORE PC LOW
210          PHA            ;
211          LDA  STRP2     ; RESTORE PC HIGH
212          PHA            ;
213          LDX  XSAV2     ; RESTORE REGISTERS
214          LDY  YSAV2     ;
215          LDA  ASAV2     ;
216 HOOK2    RTS            ; AND EXIT
217          SKP4
```

Don't worry too much just yet on what this code is doing. More details will follow later. What you want to look for is how you take something you want done, call it a big lump, a little lump, or a crumb, and then write source code for it. The source code for this portion of what you are doing should start with a title and an explanation, followed by the actual code. Lots of white space should be put wherever it will do the most good. Page breaks can be used to split things up into reasonable or logical chunks.

If we look more closely at this source code, the sequential lines become obvious. Each line has a number, label, op code, operand, and comment field, except for those lines that are pure comment and

start with a semicolon. Labels are not used in every field. But we clearly see how the NXTCHR2 label identifying line 200 is used as an operand in line 207 to find this particular line. This is how the assembler will calculate relative branches for us.

We see how operands can either be values, such as $00, or labels, such as COUT. We also see how operand labels identify locations defined or equated elsewhere in the program, such as XSAV2, which is a temporary stash for the X-register.

You can find the addressing mode used for each line by *carefully* studying the punctuation in the operand. The "#" gives us a value. STRP2 is a page zero address. COUT is an absolute address, because we happen to have previously EQU'd these to separate 8- and 16-bit address values. The parentheses in (STRP2),Y call for Apple's powerful indirect indexed addressing, which is used to hit any slot in the entire address space. We see how a label always follows a branch command, and how that label points to another nearby line in the code. The assembler can automatically find these labels, even if the source code gets shorter or longer, or if the origin of assembly changes.

As a final detail, we see three different places where operand arithmetic is used to calculate something. Where does this happen? What is going on?

Once again, the big lumps should be the highest level code you are using, handling the biggest tasks only, and delegating all details to subordinate code that follows.

After the title, you will want to add a few comment lines that explain exactly what the high level code does.

The final field of each big-lump source code should be a comment explaining what is happening. These comments should be as clear and as concise as possible.

In fact . . .

> **ALWAYS put some comment at the end of just about every op code line in your source code listing.**

There is no such things as overcommenting or overdocumenting a source code listing. Tell us at the end of each line exactly what is happening. Should you need more words than will fit a single comment line, pick up the comment on the next op code and indent an extra space or two so it looks like a continuous message.

To quote Bob Sander-Cedarlof: "The ease and neatness of comments in assembly language nearly always can make a machine language program easier to read and understand than so-called higher level languages."

LITTLE LUMPS

The little lumps are intermediate level code. These might be heavy subroutines or supporting modules. If several of these modules are needed, put them in some semblance of order, such as the order they are normally used in the program, or from complex to simple, or whatever seems reasonable at the time.

We'll skip an example of little-lumps code, since it will look just

about the same as the big-lumps code. The only difference is where it goes in the source code and how much "importance" you attach to this particular portion of your object code.

Once you get into coding your own programs, you'll find this "three-level" sort of thing becoming more and more natural and ending up as the only way to fly.

As with the big lumps, give us some comment lines at the start that tell us what is happening, and then put a comment on nearly every line of the op code source listings as well.

CRUMBS

The crumbs are what is left over after you have handled most of the code. This might be a short stash of file values or a detail subroutine or two.

Crumbs are just as important as the rest of the code. In fact, they most often end up doing practically all of the work, and most code spends most of its time down with the crumbs.

Again, we will forego an example, since a crumb to one source code might be a big lump to another. Just go through the ripoff modules and see how things are arranged.

But remember how despicable and evil structure is. Don't ever bend your program to fit the structure. Always arrange the structure to fit the program . . .

> **NEVER "bend" the program to fit the structure!**
>
> **ALWAYS "stretch" the structure to fit the program!**

In other words, stay flexible. Adjust the structure to fit the code, and not vice versa. If the structure cracks or breaks when you try this, then flush it and use something else instead. Or use super glue.

Or bailing wire.

Regardless of whether it is a big lump, a little lump, or a crumb, always provide a clear title and some explanation as to what is happening. Then show the actual code, using lots of white space and other pretty printing to make the results as pleasant to view as possible.

As a reminder, the tradition has been to use all uppercase in source codes, since some older Apples may not be able to display lowercase. But, if you are writing source code only for yourself, or only for the IIe, you can use mixed cases for your comments if you are using "new" EDASM or a "new way" word processor to enter and edit your source codes. Be sure to retain uppercase for all labels, op codes, and operands if you try this.

You can really get fancy with your borders and artwork if you use all the "new" characters on the IIe.

WORKING FILES

Practically all decent programs will use extensive data files.

A data file might be a map of rooms in an adventure, a list of notes

for a song, the text in a word processor, the template for a custom spreadsheet, a table of HIRES colors, or just about anything else you can dream up.

Stashes and working files are different . . .

> **STASH—**
>
> A very short file whose use is not often changed that usually immediately follows its controlling code.
>
> **WORKING FILE—**
>
> A usually long file that normally goes at the end of a program and whose use may change with program needs.
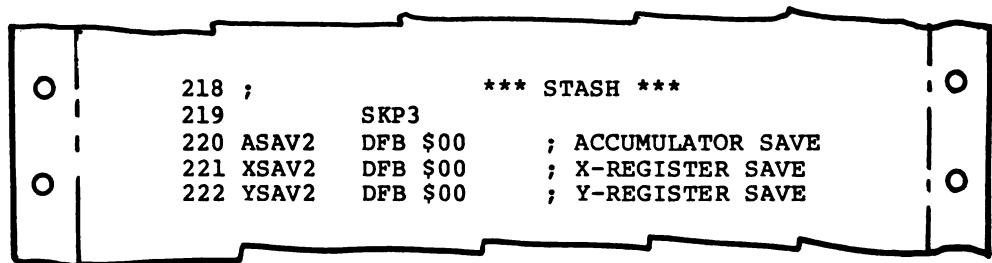
As specific examples, in an adventure program, the message "IT IS TOO DARK TO SEE" would be a stash that immediately follows the code that checks to see if a lamp or torch is still lit in a dark room.

On the other hand, if you know what you are doing, you can use *one* main program to handle many different adventures, just by changing the working files that get tacked onto the end.

For instance, all of *Scott Adam's Adventures* 1-12 use exactly the same controlling code. Only the data files get changed to alter the situation and the responses. Infocom Inc. uses nearly the same program for *Zork I, Zork II, Zork III, Starcross, Infidel, Planetfall,* and *Deadline.*

Working files should follow the rest of the code for several reasons. They are simplest to change this way. In addition, an assembler's editor may not be the best way to generate long files. Instead, a word processor or a custom code generator might often be better choices. For instance, you might use Applesloth to generate a sine or cosine table for trig uses. With the main files at the end, you can combine assembler source code with word processor files, or use any other combination that works for you.

Here's an example of a *stash* . . .

```
218 ;                        *** STASH ***
219        SKP3
220 ASAV2  DFB $00   ; ACCUMULATOR SAVE
221 XSAV2  DFB $00   ; X-REGISTER SAVE
222 YSAV2  DFB $00   ; Y-REGISTER SAVE
```

This stash is 3 bytes long, and is used to set aside a "safe" area to save and then restore register values. In this particular case, we only set aside the storage by using the "dummy" $00 values. These will later get replaced with real saves as the program is run. In other stashes, you will carefully and exactly define what goes into a location ahead of time.

In this case, the stash is the only one used, and its use is obvious when it comes up in the assembly listing. Normally, though, you will

want to have a few comment lines that explain exactly what the stash is up to.

As usual, tell us what you have, tell us what it does, and then add comments to each and every line.

Yes, stashes are really crumbs, and normally are tacked onto the end of crumb code. But remember that a stash is a file that holds something, rather than working code. You *run* code, but you *use* stashes.

Working files are definitely not stashes. They are blocks of data values you place near the end of your object code. The length and nature of your bulk files change, of course, with your intended program goals.

Relatively short work files can be created with the assembler with the DFB command to define one or a few bytes at a time. For best appearances on the assembler listing, you should define only 8 or fewer bytes per line. You are allowed to "pad" three spaces between your DFB and the data values. This will move all the data values over into the comment field for improved appearance.

As a long file example, here's part of a HIRES pattern file used to get the 191 HIRES fast background colors in *Enhancing Your Apple II* (Sams 21822) . . .

```
165 ;              *** COLOR PATTERN FILE ***
166        SKP2
167        ORG   COLOR+$100
168        SKP2
169 CFILE  DFB      $00,$00,$00,$00,$00,$00,$00,$00
170 PAT1   DFB      $2A,$55,$2A,$55,$2A,$55,$2A,$55
171 PAT2   DFB      $D5,$AA,$D5,$AA,$D5,$AA,$D5,$AA
172 PAT3   DFB      $7F,$7F,$7F,$7F,$7F,$7F,$7F,$7F
173 PAT4   DFB      $80,$80,$80,$80,$80,$80,$80,$80
```

In this case, 8 bytes are used at once to set the background color of the HIRES display cell that is 4 bytes wide and two scan lines high. The file continues for a total of 256 bytes. The file holds a total of 32 of the possible 191 solid HIRES colors or your choice of any of a mind-boggling 18,446,744,073,709,551,616 HIRES patterns.

Files can be any length you need, but their access gets trickier if you go beyond 256 bytes. Files of a page or less can be reached with indexed addressing, while you have to go to indirect indexed addressing for longer entries.
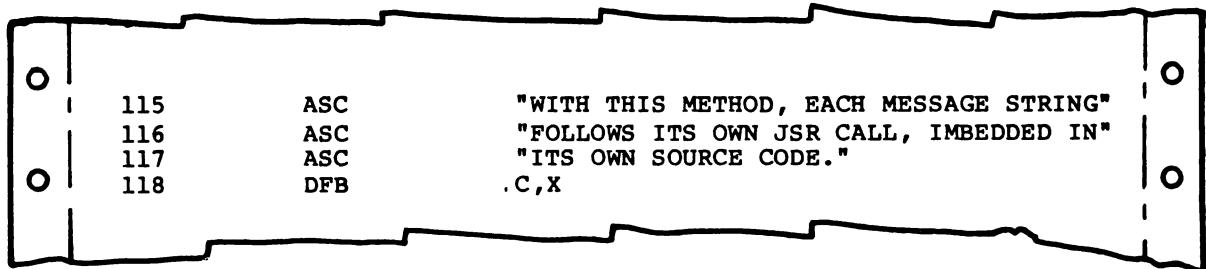
If your file meaning and intended use is not totally obvious, be sure to add a few comment lines at the beginning to explain what is happening. Do this the same way you did the documentation on the big lumps, little lumps, and crumbs.

You don't have to use DFBs to create your file if you don't want to. Sometimes, you can use DWs to define 16-bit addresses in a file, or DDBs to define other 16-bit values that are to appear in "frontward" order. But, note that you only are allowed one DW or DDB per source code line. We've seen how you can have up to eight DFBs per line instead.

And, once again, there may be better ways to build long working

files, such as using a word processor, a custom program, a graphics tablet, input from a scanning plotter, digitized video, or whatever.

You can also use the ASC command to generate ASCII message files . . .

```
115        ASC        "WITH THIS METHOD, EACH MESSAGE STRING"
116        ASC        "FOLLOWS ITS OWN JSR CALL, IMBEDDED IN"
117        ASC        "ITS OWN SOURCE CODE."
118        DFB        .C,X
```

In this case, we have used the ASC pseudo-op to generate our text, and the DFB pseudo-op to handle a carriage return and a "double zero" end-of-message marker. This happens only because the source code previously EQU'd the label C to $8D and an X to $00. You'll find several examples of this in the ripoff modules.

Remember that the ASC command always starts with a delimiter. This can usually be a quote symbol. If you actually want a quote symbol to appear on the screen, you should use some other symbol instead.

Like so . . .

```
ASC   "ITS TOO DARK TO SEE"

ASC   /PRESS "RETURN" TO CONTINUE/
```

Again as a reminder, the first delimiter is essential, but the end one can be left off if there are no comments added to this line. But this gets dangerous if there are trailing spaces. What you see might not be what you get.

Summing up . . .

> **Limit each individual ASC command pseudo-op to 40 or fewer characters.**
>
> **Put three spaces between ASC and the first delimiter of your ASCII message.**
>
> **Do control commands with separate DFBs using simplified labels.**

These hints will give you a very clean printout on your assembler listing, and make for easy insertion of control codes.

"New" EDASM also has a STR command that precedes your text string with a character count byte.

Creative use of labels can let your assembler do most of the dogwork of converting from "people" to "machine" values. Note how much easier it is to enter a DFB C than it is to look up the ASCII carriage return value and then do a DFB $8D. It gets even better when several different control commands are handled by one DFB.

The "old" EDASM editor does not directly support lower case ASCII messages, but "new" EDASM and both its assemblers do. If you are

editing with a "new way" word processor, it is a trivial matter to enter lower case ASCII into files. Just be sure you use the CAPS LOCK on all your labels, op codes, and for all operands other than ASC or STR.

Remember that lower case characters will show up as gibberish on an unmodified Apple II or II+ screen, unless you convert them back to upper case before displaying them. To be compatible both ways, you insert a "DO YOU HAVE LOWERCASE?" prompt inside your program, and then make some corrections if lower case is not available. "New" EDASM gives you several ways to handle this.

Most newer assemblers can directly generate lower case text. If yours cannot, just use a word processor for the editor, or else take the final object code and "force feed" case changes where you want them.
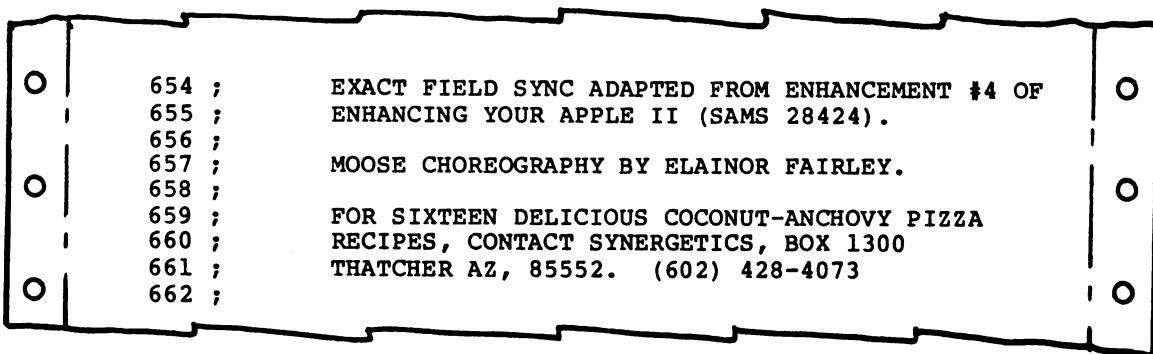
A word processor is usually the best way to generate text messages, particularly if they are very long or involve lower case. Instead of using ASC pseudo-ops, you build an entire message file with your word processor, binary save it to disk, and then link that message file directly with your object code. This method entirely bypasses all assembly hassles, although a text file to binary file conversion routine might be needed.

BOTTOM LINE COMMENTS

At this point, your source code should already have all the information the assembler needs to put together object code. If anything seems to be missing, go back and stuff it into its proper big lumps, little lumps, crumbs, or working file slot.

The bottom line is a good place to put credits and any other notes or comments you feel you really should mention.

Here are some typical bottom line entries . . .

```
654 ;    EXACT FIELD SYNC ADAPTED FROM ENHANCEMENT #4 OF
655 ;    ENHANCING YOUR APPLE II (SAMS 28424).
656 ;
657 ;    MOOSE CHOREOGRAPHY BY ELAINOR FAIRLEY.
658 ;
659 ;    FOR SIXTEEN DELICIOUS COCONUT-ANCHOVY PIZZA
660 ;    RECIPES, CONTACT SYNERGETICS, BOX 1300
661 ;    THATCHER AZ, 85552.  (602) 428-4073
662 ;
```

The bottom line is where you give credit for what was done and a sales pitch for anything newer or better that you may offer. Put anything you want to here. Just don't preach to the choir.

LABEL REFERENCES

Strictly speaking, our final two parts to our structure aren't really part of the source code. Instead, they are things that get tacked onto the end of the assembly listing during the assembly process.

Anyway . . .

One very handy thing to have when you are analyzing any machine language program is a list of who does what to whom. Most assem-

blers will provide some sort of a reference list at the end of their listing if you ask them to.

EDASM provides a simple version called a symbol list or a *label list*. Other assemblers or disassemblers may provide a more complex listing called a *cross reference* . . .

LABEL LIST—

A listing of each and every label used in a source code, along with the label address or label value.

CROSS REFERENCE—

A listing of each accessed memory address, along with the addresses of who called it for a load, jump, store, branch, subroutine call, or whatever.

The label listing shows you which labels get used. They also point out glaring errors, such as reusing the same label, or getting an op code such as CLC, over in the label column by mistake. Improperly spelled labels or labels "alike but different somehow" become obvious. Really strange labels that result from typing errors also will leap out at you from the label list.

EDASM flags any unused labels with question marks. This brings these to your attention, just in case the label really is a mistake, rather than something that you put in as a memory jogger or for outside access.

The EDASM label listing is normally provided in alphabetical order by label spelling, and then in numeric order by label value. These listings are extremely handy when you need them, but otherwise are a painful waste of printer time and paper. You can turn the EDASM label listing on or off by a LST OFF or LST ON command as the *last* line in your source code.

"New" EDASM gives you all sorts of "mix and match" listing options, including execution time display and several formatting options. See Appendix A.

A full cross reference is much more powerful than a simple label listing, since these show you every use of every address used or referred to anywhere in the program. This is most useful for tearing into other people's programs, and for answering, "Now why on earth did I do that?" on your own source codes. Full cross references are often broken down into separate internal, external, and page zero listings.

Some other assemblers will provide a full cross reference on command. The simplest way to get a complete cross reference out of EDASM is to assemble your object code and then disassemble it with *Rak-Ware's* DISASM or something similar.

ERROR MESSAGES

One of the first joys you will undoubtedly encounter on an early assembly attempt, is that you will write a ten line program and get 34 error messages as a result.

You get error messages when you tell the assembler to do something so stupid that it simply doesn't understand you . . .

> **ERROR MESSAGE—**
>
> **A listing of a problem you created by telling the assembler to do something illegal or something it just plain did not understand.**

EDASM generates two different kinds of error messages, those that apply to the assembly process, and those that involve disk access.

We'll look at specific error messages later. Our goal here is to recognize that you'll most likely get bunches of error messages tacked onto the end of your assembler listing.

Some errors are very subtle. Mixing up "eyes," "els," and "ones" or "ohs" and "zeros" are good ways to confuse EDASM. Having too many or too few tabs or tabbing spaces so you get in the wrong field is another. This one gives you a label called LDA or an op code named RESTART.

Of course, leaving that "$" sign off an operand will automatically mix up decimal and hex values. An entry of LDA #60 will put a value of #3C in the accumulator. And, of course, an entry of LDA 60 can put darn near anything in the accumulator. Why?

An extra comma at the end of a string of DFBs does all sorts of nasty things, as does forgetting or imbedding the delimiter in an ASCII string.

Blank lines in EDASM are a no-no. Yet, they are *absolutely essential* for a decent and readable assembly listing. Use the SKP pseudo-op instead, or else a comment line that starts with ";" or "*." It is very easy to get fumble fingered or hit the wrong keys to exit an "old way" editing or entry mode. This puts strange values onto strange program lines for you. It is also easy to accidentally imbed control characters inside your program lines during "old way" editing.

Labels and other values default to zero if not defined. Which creates bunches of new problems all by itself. But, you'll find out all of this soon enough on your own.

Will you ever.

There is one totally meaningless error message that EDASM may give you sometime. This message is "***SUCCESSFUL ASSEMBLY: NO ERRORS."

All this message tells you is that . . .

> **A hard-copy printout that says**
>
> **SUCCESSFUL ASSEMBLY: NO ERRORS**
>
> **tells you that:**
>
>   1. **The printer was turned on,**
>   2. **There was enough paper,**
>   3. **The ribbon is not illegible.**

You see, it is perfectly possible to enter totally legal commands in totally legal sequences to EDASM, and still end up with object code that simply won't work. All the success message tells you is that you haven't done something so blatantly stupid that EDASM couldn't take a stab at guessing what it was you were trying to tell it to do.

Naturally, if you have error messages, the program is *guaranteed* not to run. If you have a success message, the program probably will *still* not run, or at least not on your first dozen tries.

For a successful program, you must first enter and use legal commands in a legal way in EDASM. After that, the object code that EDASM generates also must do legal things in an expected way for your program to work.

That just about completes our sixteen steps of a structured assembly language listing. The first of the ripoff modules is called EMPTY SHELL.SOURCE, and appears later. This one is sort of like a Christmas tree you can hang all your ornaments on, and greatly simplifies organizing and structuring your source code. Since it is usually easier to edit a source code than create one, you might like to start with the empty shell and adapt it to your particular needs. Much more on this later.

For now, though, it looks like we are almost ready to actually enter and edit some source code.

DIFFERENCES BETWEEN MY STRUCTURE AND YOURS:

4

WRITING AND EDITING
SOURCE CODE
(the OLD way)

There are at least two possible ways to write and edit your source code. The "old way" involves using the editor part of the assembly language development system. The "new way" uses a word processor along with a supervisory control language instead. We'll cover the old way in this chapter and the new way in the next. Be sure you read this chapter, and practice and understand the old way before you try the new.

OK. We now know all about source code lines and know at least one evil and despicable way to structure source code. But, where does that get us?

You turn on the assembler, and there's an insidious colon staring at you. What now? One thing you can do is load up the EMPTY SHELL.SOURCE ripoff module, and that gets you several hundred lines of source code right off the bat. Of course, the empty shell won't do what you want it to. Instead, you somehow have to *edit* the empty shell into something useful that works for you.

Where do you really start?

> **You write assembler source code in exactly the same way you already hand wrote and hand debugged your own machine language programs.**

Remember that an assembler will *not* write programs for you. All an

assembler can do is greatly ease and simplify the process of creating machine language code. So, if you have not hand written and hand debugged a few hundred lines of machine language code, an assembler is totally useless.

Dangerous, even.

Use whatever methods that already worked for you when you did your machine language programming. There is a sequence called the *fourteen steps* that appears in *Don Lancaster's Micro Cookbooks* (Sams 21828, 21829). The fourteen steps show you how to go about attacking and solving real world microcomputer problems, and lend themselves beautifully to Apple II or IIe use of EDASM.

Use the fourteen steps if you like, or else use whatever methods you already have on hand and have used for your already written and debugged machine language programs. But, please don't go beyond this point unless you have the fundamentals down solid . . .

> **DO NOT GO PAST THIS POINT IN THIS BOOK UNLESS YOU ARE ALREADY A COMPETENT MACHINE LANGUAGE PROGRAMMER!**

Uh, there's all those turkey feathers again. Oh well, That should be the last of them. Before we turn to the exact mechanics of writing your own source code, let's take a brief look at . . .

PROGRAM STYLE

There are many different 6502 op codes available, and lots of different ways you can arrange these into a working program. Which ones should you use? In what order should you put them?

Questions like these involve program *style* . . .

> **PROGRAM STYLE—**
>
> The choice of op codes used and the order they appear in a machine language or other program.

Program style also includes your approach to a problem and how you arrange the things you think are important. Style obviously influences the overall program vibes and decides how elegantly or how simply the job gets done.

Or how well.

Let's look at some factors that affect program style, and see how these factors force a certain flavor to the style of any program you happen to be writing.

Speed

Most any old machine language program assembled by a more or less competent programmer can run the wheels off most any program compiled from a higher level language, and will pass a program that is

interpreted from a higher level language like it was sitting up on blocks.

As we've seen, the reasons are that machine compilers have to handle all possible things all possible ways, and are thus inherently slower, dumber, and less efficient than what can be done creatively by a decent machine language programmer working toward a specific solution. Interpreters, of course, are much slower than compilers, since each high level instruction has to be individually converted into its own machine language code before it can be executed.

But, beyond this, there is fast, faster, and there it went.

Extreme program speed gets important in animation, text processing, graphics manipulation, multiplication, 3-D perspective, direct external control, spreadsheets, business sorts, and other applications like these.

So, if you need extreme speed, you go to extreme measures, over and above the normal speedup that a normal machine language program can give you.

For extreme speeds, your style should include mostly *straight line* coding, in which each instruction is done individually with the fastest possible op codes. Loops should be avoided in those parts of the code that have to run super fast. If you must use a loop, share the loop overhead as many different ways as you can by having whatever is inside the loop do as many different things as possible.

Subroutines are also a no-no in those parts of the code that must run very fast. The reason for this is that each subroutine call and return needs a time overhead of 12 CPU cycles, or roughly 12 microseconds.

One trick that leads to code speedup is called *table lookup* . . .

> **TABLE LOOKUP—**
>
> **A programming method where you get needed values out of a pre-defined table, rather than by calculation.**

One of the most dramatic places to use table lookup involves calculating text, LORES, or HIRES base addresses. There is a tremendous speedup involved in getting the base addresses directly out of a table, rather than calculating them.

Table lookup, though, almost always will take up much more memory space than will calculation methods. Should the space get totally out of hand, sometimes you can split the table into smaller parts. For instance, an 8*8 multiply table would be horrendous, but simple factoring of (A+B)*(C+D) and four trips through a 4*4 multiply table will still be reasonably fast and need only a page of memory.

As a plum for you math freaks, the fastest software multiply scheme I know of at this writing uses factoring of $2XY = X^2 + Y^2 - |X-Y|^2$. Only 512 bytes of "(X*Y)/2" table lookup are needed for a very fast 8 X 8 multiply.

Another speedup trick is called *offloading*. When offloading, you try to handle anything slow with anything *but* the CPU.

For instance, you can use an external UART or serial interface chip to generate its own serial code. This way, the CPU only has to pass the

command to the serial device, and then the serial device can take its own good time outputting the code. The CPU then is free to do other things while the code is being output.

Other examples of offloading include a printer buffer. Here, the CPU passes a long message to external RAM. The external RAM then holds the message long enough that a slow printer can output the message.

Timers and real-time clocks can also handle long timing tasks independently of the CPU. Interrupts can also be handy, since the CPU can be doing useful things instead of just polling a keyboard or otherwise waiting around for something to happen. External music chips or sound effects generators are also useful offloaders, in that a very few CPU commands can quickly activate them.

Should you have some very fancy calculations to make, consider adding a trig chip or a fast multiplier or floating point chip to your system. Here the CPU passes the problem to the external chip, and the chip passes the answer back to the CPU. Even if a slow old trig calculator chip is used, the CPU can go ahead and do other things and then come back later for an answer.

Watching how things get where they are supposed to go will also speed things up bunches. Obviously, you can output stuff much faster at 9600 baud than at 150. Faster still with a parallel port. And you can get to the screen much quicker if you do not waltz your way through DOS, a printer card, an echo, and long monitor routines on the way. As another example, any time the screen scrolls, things shut down for many milliseconds. So, avoid scrolling during time-critical events.

A mix of hardware and software working together will almost always do some task quicker, cheaper, and much faster than hardware or software can standing alone. So, it always pays to think both in terms of hardware and software, and not just one or the other.

Another key rule to speeding up programs is to never attack something head on. The "obvious" solution, done the way that "they" want you to, is *never* the fastest or best approach. Always try something off the wall and see where it will lead you.

Ferinstance, the HIRES graphics routines in Applesoft were optimized for *length* rather than speed. Since Applesoft is so slow anyhow, it didn't matter. At the time, a 16K Apple was a biggy, so space was all important. Your own assembly language code can speed HIRES animation by great heaping bunches. Yet, there *still* are commercial programs that even today persist in using these intentionally slow subroutines.

The usual penalties paid for speed in a program are that the program will often get much longer and take up much more room in memory. Extreme speeds also add to the programmer's time and creative effort.

Summing up . . .

```
┌─────────────────────────────────────┐
│                                     │
│          FOR MAXIMUM SPEED—         │
│                                     │
├─────────────────────────────────────┤
│                                     │
│  Use straight line code.            │
│  Eliminate or share loops.          │
│  Avoid using subroutines.           │
│  Look up, rather than calculate.    │
│                                     │
│  Offload time-consuming tasks.      │
│  Mix hardware and software.         │
│  Never attack head on.              │
│  Try an off-the-wall approach.      │
│                                     │
└─────────────────────────────────────┘
```

Normally, speed for the sake of speed itself is not a good idea, since your code may get excessively long, besides being harder to write and understand. You have to balance speed against . . .

Length

It has been proven time and time again that programs that seemed to demand all of the available memory space of a dino or minicomputer can easily be shoved into an Apple II or IIe with room to spare.

But, to do this, you must use one or more code shortening tricks that influence your program style.

Loops and subroutines are your foremost two tools for shortening code. Although not normally a good programming idea, multiple entry points to loops and subs can dramatically shorten your code.

A loop shortens things by using the same code over and over again for some number of times, or until some result happens. A subroutine shortens code by being accessed from many different places in a program.

Another way to shorten code is to use the most powerful and most elegant instructions available. The indirect indexed command in the 6502 can be used to access most anything most anywhere in the most general and most flexible manner. Learning these super-powerful commands is a must if you are to minimize code space.

Going to a custom DOS that only boots, or else does only what needs to be done, is another code shortening trick. But, please keep what the DOS does standard and do it in a standard way.

Overwriting memory is yet another space saving trick. For instance, on a HIRES game program that only has to boot, put the DOS and any setup code on the HIRES pages. After booting and setup, these pages then become the game playing field.

Finding nooks and crannies can also help bunches, if you have a program that almost, but not quite, fits. There are parts of page zero, the bottom of page one, the top of page two, most of the lower page three available for special uses, as are a few random locations "hidden" on the text and HIRES pages. It's also possible to stuff things into DOS buffers for special uses. RAM cards can help bunches on older Apples.

The IIe gives you a whole new ball game with its 128K of RAM and lots of ways to expand for even more RAM.

Virtual memory that involves repeated disk access is one way to extend the Apple's memory to include everything you can put on two diskettes in two drives. Even more if you want. You can also swap things back and forth to add-on memory cards, or simply off-load DOS onto a 16K memory card.

Newer versions of DOS offer dramatic speedup of disk access, so the grinding and whirring of the disk during use is now greatly minimized. In fact, the grinding and whirring can be eliminated almost entirely by going to a RAM disk emulator.

One very powerful way to shorten code is to create your own interpreter. Part of your program now becomes a machine language way to interpret brief instructions from a master file.

Using an interpreter, of course, is how Applesoft and Integer BASIC work. But special interpreters appear in many other programs. *Zork* uses an interpreter for MDL, which is a subset of LISP with Icelandic subtitles. *Adam's Adventures* have their own special interpreter. Such programs as the graphics language GRAFORTH, or the artist's language CEEMAC, also use special interpreters.

Special compaction codes are another possibility. ASCII text is only 25 percent efficient at storing normal English text, and can be even poorer than this if you know ahead of time exactly what the text is. There are all sorts of text compaction schemes available. These include the "three characters in two bytes" used by *Zork*, the "character pair" method used by Adventure International, the "matched text filter" method by Synergetics, and the "changes only" method used by most spelling programs and classic cell animation.

But, don't get sucked into using *Huffman* codes as text compactors. These are nearly worthless for real-world micro text compaction, due to their limited benefits and variable bit lengths. There are many better alternatives available that are more micro-oriented. Huffman codes do have one outstanding use. They make great university level student paper topics.

Data compaction is another possibility. For instance, a normal HIRES picture takes up 34 diskette sectors. If you remember to throw away the last unused 8 bytes, you can slash this clear down to 33 sectors.

But it takes up much less space if you encode the picture some other way. You can do this by using shape tables, by using HIRES 8 X 8 specially defined character blocks, by saving changes only, by having a special interpreter that creates images from brief instructions, or by using sprites that can be mapped and removed from existing HIRES scenes. There are many other possibilities.

In general, it should be possible to store a LORES image in 100 bytes and a HIRES image in 1000 bytes with decent compaction tricks. This translates to 130 HIRES pictures or 1300-odd LORES pictures per diskette.

Music files can be shortened by precoding so that the pitch and duration need only a single byte. This results in a 2:1 compaction.

Files of names and addresses can be shortened by having a brief, sortable file in RAM that points to a main file on disk. In the brief file, you only put the stuff you want to sort against. One version of this is called ISAM, which stands for the *indexed sequential access method*.

There are more subtle ways of squashing file information. For instance, if all of your customers reside in one of ten zip code areas,

you need only store a single 0–9 byte, instead of the five needed for the full zip code. Use table lookup to find the right code. Even if only most of your customers are within nine zips, you can create a small "exception file" for those who aren't.

Similarly, if most of your customers live in one of nine towns, use a single digit 1–9 as a town pointer in a data file, and a 0 as an exception pointer. This also eliminates the need to put the state into each individual file.

The general idea is to put the most compact pointers you can inside each record of a long file, and then use those compact pointers to table lookup the actual values needed.

A largely unexplored trick called *run length encoding* can also be applied to speech synthesis and HIRES displays, where the code tells you to make the next so many bytes a certain pattern. This exciting new concept has so far seen little micro use.

Taken to its extreme, run length encoding lets you add a very fast RAM to the *video* output of your Apple. You can feed commands to this fast RAM at a *seven-million* bit-per-second rate (!) with your stock Apple. This means there is no reasonable limit to the Apple display colors, resolution, or gray scale at the output of this fast RAM.

Or why video at all? What could you do with a magic pipe that spits out ones and zeros at a seven-megabaud peak data rate?

Heavy.

Rethinking the program is one good way to shorten it. What are the features that use up the most space? Are they really needed? Can they be combined into some other form? Can you trade off a little performance loss for a lot of space saving?

Ironically, two key rules that speed up programs also will shorten them. These, of course, are not to attack anything head on and to try an off-the-wall approach.

To sum up the style tricks involved in code shortening . . .

FOR MINIMUM SIZE

Use loops and subroutines.
Choose powerful addressing modes.
Try a custom, shorter DOS.
Overwrite memory.

Don't overlook nooks and crannies.
Create a custom interpreter.
Employ virtual disk memory.
Compact text files.

Pack pictures and graphics.
Run length encode.
Rethink the program goals.
Try an off-the-wall approach.

The obvious disadvantages of shortening programs are that they may run slower, will take longer to write and debug, and can become incomprehensibly complicated to understand.

Other Style Factors

Speed and program size are the two biggest factors that influence your program style. But there are also lesser things that influence your programming style or add flavor to your assembler techniques. Let's take a quick look at some of these.

The time it takes you to write a program can be important. In fact, if you are after a quick and simple "use once" result, you may even be better off using BASIC. If you want to minimize your program writing time, it pays to stick to mainstream constructs and to freely adapt what others already have done. Naturally, you'll not only miss the brass ring this way, but you won't even be along for the ride.

Actually, what looks like an awful lot of work usually only has to be done once. Your early assembler work is very much a learning process that you should be able to build on. Avoid the work at the beginning, and you *never* will become a decent assembly language programmer.

Guaranteed.

Once you start actually developing useful programs, you'll have techniques and ready-to-go modules you can work with and build on. In theory, you can get from a cold start to being a decent and useful assembly language programmer in less than three years of solid effort. But this has not happened yet, not even once.

Taint likely, either. So . . .

> **Don't be alarmed at the time and effort you have to put into writing an assembly language program.**
>
> **Much of this time is part of a learning experience that you will not have to duplicate later.**
>
> **Other effort will create modules that you can build and use in the future.**

There is a very quaint concept that seems to surround personal computer programming. This silly concept believes that your customers are to pay you for your learning experiences and all the mistakes you made along the way. If this was done with book authorship, a similar idea would say that you should pay much *more* for poorly written work by an inexperienced author, simply since it took him such a long time to do.

Putting a stiff price on a program just because you spent a year on it is patently absurd, if 51 of the weeks in that year were part of a learning process, involved stupid mistakes, or otherwise resulted in the usual bumbling incompetence that is painfully obvious in well over 95 percent of all available Apple software.

The key question should not be how long it took you to write a program, but how long it would take a competent and knowing programmer to rewrite and redo the code from scratch, knowing what you *should* know by now, and using the tools you *should* now have on hand.

Another question of style involves relocatability. Most typical machine language programs will operate properly from only one posi-

tion in memory and then only from a valid starting point. If you want the program to run anywhere or on any machine, you have to go to a lot of trouble to make things *relocatable*.

The simplest thing you can do to build a relocatable program is to use no jumps, no subroutine calls, and no internal references, handling everything instead with relative branch commands. This type of program will run anywhere in memory without any hassles. The penalty, of course, is that this program style is extremely limiting.

Dullsville even. Like the "three B's" of *all* educational software to date—bad, banal, and boring.

The next step up in relocation is to use your assembler to relocate the program for you, simply by changing the ORG command, and being sure you used a label every place an absolute address might crop up. The result of this still will be a program that can only run in one place, but you can relocate everything to suit the place that is needed.

Yet another route is to tack a custom relocator onto the front of your program. This custom relocator finds out where the program is sitting and then generates a base address table of some sort. This base address table is then used to modify locations in the code so that things will run in their present position.

EDASM also provides for fully relocatable code assembly by using the DOS "R" file format and a loader such as LOADHRCG. This route is complicated and specialized, and there are use problems. Bugs even. One particular hassle with LOADHRCG is that the pointers move down in memory each booting, eventually plowing all of RAM. Another is that the Applesoft string pointers do not get properly reset.

The heaviest way of all to relocate a program is with a cross assembler and an emulator. This lets you transfer programs to totally different computer systems.

Summing up . . .

WAYS TO RELOCATE CODE
Use relative code only. Use EDASM with a new ORG. Use a location finder and adjuster.
Use "R" files and "R" assembly. Use a linking loader. Use a cross assembler or emulator.

Another style problem involves whether your program will stand alone or be part of a larger program. If the module you are working on is to stand alone, then there are no limits to the labels you use.

On the other hand, if the module has to work with other modules, you either have to use an assembler that has both global and local variable capabilities, or else you have to make sure that each label is unique.

We've used a numeral after many of labels in the *ripoff modules* so that all of the modules have similar, yet separable, labels. To reuse the label, just change the end numeral.

The rule here is simple enough . . .

> **With an assembler that does not allow global and local variables, all labels on all modules must be unique.**

A local label capability is included in "new" EDASM's macros. "Old" EDASM did not provide this.

Our next style question involves *self-modifying* code . . .

> **SELF-MODIFYING CODE—**
>
> **A computer program that modifies itself as it goes along.**
>
> **Such code is extremely powerful, highly dangerous, and the parts to be modified must reside in RAM.**

A self-modifying program changes some memory locations as it goes along. Actually, just about any program changes some memory locations as it is used. The degree of self-modification is what is critical. If the program changes its *own* working code, rather than just modifying constants, or file values, you have just opened a very large can of worms.

Self-modifying code can be extremely fast and extremely compact, and literally can leap tall buildings in a single bound. It can also destroy itself and everything else in the machine, in the disk drive, or even nearby. Self-modifying code can also box itself into a corner from which there is no escape.

The best rule on self-modifying code is . . .

> **NEVER use self-modifying code, unless you know exactly what you are doing!**

Something that goes along with this involves what kind of memory the program will sit in. Most programs run in RAM and are thus allowed to change any of their locations. If the program must run in ROM, though, no ROM code location is ever allowed to change.

You might run into "must sit in ROM" code in designing your own plug-in cards, in substituting for the Apple II or IIe ROMs, in code that must boot on a cold Apple, or in using the Apple II or IIe as an emulator for a controller or some other dedicated micro use where the code must be permanently burned into EPROM.

Compatibility is another style influencer. Does your program have to interact with DOS, the system monitor, Applesoft, Integer, or some other language? If so, there are very specific restrictions on where you *put your program* module, how you pass information to and from it, which page zero locations you are allowed to use, and how you interact with the host language.

A final important style modifier involves the smarts of your user. If your user is inexperienced, your program should be self-prompting,

self-explaining, and should have a built-in tutorial of what the program is and how it is supposed to work.

Any decent program should be self-documenting and self-explaining. At the very least, your error trapping should successfully handle a pussycat chasing a lizard across the keyboard. Or, for that matter, a lizard chasing a pussycat.

The key rule here is . . .

> **ALWAYS assume your user is an epsilon minus, but NEVER insult his intelligence.**

To paraphrase Murphy, anything that can go wrong with a program will go wrong, one time or another. And, even if the program is perfect, things will still go wrong should a plug-in card get dirty, or should DOS foul up one way or another, or an air conditioner surge the power line.

The degree of error trapping and hand holding you provide should depend on the value of the program to a user. A program that holds a year's worth of general ledger receipts, or one that is interacting with some life support equipment obviously needs more error trapping than a low cost game does.

But, I seriously doubt if you can ever provide too much documentation, tutorial help, or error trapping in any program you may ever write.

UNSTYLE

Just as there are elements of style to any assembly language program you may write, you can blow it all by doing something stupid that totally turns your customers off.

We'll call *unstyle* anything that damages what you are trying to accomplish.

The dumbest, stupidest, unstyle move you can possibly make is to lock your program. This only hacks off your legitimate customers and throws an open challenge out to the uncopy buffs.

In fact . . .

> **There is no such thing as a "locked" or "protected" diskette program.**
>
> **All locking does is delay very slightly the opening of your program and enhance greatly the incentive to do so.**

The second most stupidest thing you can do is use special DOS files that are not readable by the rest of the world. Any program that involves itself with creating any files at all that are business related should be readable by standard means. This usually means using standard DOS text files, or else using standard and thoroughly documented data formats, such as DIF, or something similar.

The third most stupidest thing you can do is not make your source code available at nominal cost for the asking. The really great things

that will get done with your program are things that you haven't even thought about and don't even suspect. The true value of a program comes about only *after* others can interact with it and then modify that program to suit their own needs and new uses.

The fourth most stupidest thing you can do is fail to support your users. At the very least, users reasonably should be able to expect instant and free unlimited backup copies, a totally free defective diskette exchange for six months after the sale, and at least one year's totally free consulting via a telephone hotline.

Another very important part of user support is proper *beta* testing of your programs. In beta testing, a group of controlled but disinterested outsiders thoroughly tests and evaluates your software to find the most blatant bugs and use problems. Users will invariably have different thought processes and needs than those of the author. Only through thorough, pre-release beta testing can these differences come to light.

The fifth most stupidest thing you can do is make promises, excuses, apologies, or explanations aimed at weaseling your way out of some second-, third-, or ninety-ninth-rate feature of your work. Like promising a part that does not exist. Or using a nonstandard keyboard. Or claiming it is possible to read dot-matrix print. Or using "preliminary" documentation. Or requiring hours for a sort. Or loosing data files. Or leading with your ego. Or getting defensive over a just plain stupid mistake.

Or an obvious oversight.

The absolute key feature of the Apple IIe is that it eliminated in one swell foop all the promises, excuses, apologies, and explanations that the Apple II needed.

Such as short screens, no lowercase, difficult memory expansion, tricky keypad add-ons, unreliable switches, kamikaze card changes, nonstandard keyboards, and tricky paddle access. Similarly, the key difference between old versions of *Apple Writer*, and the new IIe version is that there are no promises, excuses, apologies, or explanations needed. The thing just works beautifully, and is far and away the *only* programmable word processor that is genuinely lots of fun to use.

The sixth most stupidest thing you can do is to skimp on the documentation. Well over half your effort should go into the manual and the package that goes with your diskette. In fact, this is the ultimate copy protector. What you simply do is make the contents of your diskette a small portion of the total package you are offering. Put the value added into the documentation, support services, and any companion hardware and you are home free.

Paperwork that goes with your program should include a tutorial, an index, a pocket card, and hotline help. People who write documentation on a dot-matrix printer should be staked to the nearest anthill, and left there until the next meeting of the steering committee.

The seventh most stupidest thing you can do is to make your program fragile. A fragile program is one that self-destructs at an early date. Any speed-sensitive disk access scheme *guarantees* fragility. For that matter, you should *never* rewrite to your object code disk. This is what makes *Castle Wolfenstein* ludicrously fragile, despite its otherwise excellent features.

Never write to a game diskette!

Other sure routes toward fragility are to forget to initialize things properly, or not fully test your code under *all* possible use conditions,

or to attempt to interact with a higher level language without thoroughly understanding what is going on. Or to ignore *all* possible use configurations.

The eighth most stupidest thing you can possibly do with your program is overprice it. There never has been, and probably never will be, an Apple II computer program worth over $24.99. Any difference between this *value* and the charged *price* of a program is immediately offset by the number of bootleg copies in existence. This is a perfect example of supply and demand economics.

The ninth most stupidest thing you can possibly do with your program is to use deceptive cover art and promotion. If your HIRES graphics are second rate, me-too imitations, don't put a pulp novel cover on the package showing things differently. If you are ashamed to put the actual graphics the program uses on the cover, then leave the cover off entirely. If your program is a ninth-generation misquote of *Hammurabi* or *Eamon,* then say so.

Let's sum up unstyle . . .

**THERE IS NO POINT IN WRITING
AN ASSEMBLER PROGRAM IF**

You lock it.
You use special DOS.
You don't provide source code.

You fail to support your users.
You make excuses and promises.
You skimp on documentation.

You create a fragile program.
You overprice it.
You use deceptive art and ads.

Above all, if you can't accept any of the previous unstyle concepts, please go away. I definitely do not want you using any of the information in this book to perpetuate any of these absurd practices.

Now, we finally should be ready to begin writing our source code. Let's get into the mechanics of . . .

WRITING "OLD WAY" SOURCE CODE

Once again, there's an "old way" and a "new way" to handle source code. The old way involves using the editor portion of your editor/assembly package in the intended manner. Be sure to understand what is supposed to happen and how it is supposed to work before you try anything else.

We'll continue using EDASM here for our examples. As before, if you have a different assembler, use page highlighters, margin notes, and the end-of-chapter boxes to show any differences between your assembler and EDASM.

Once again, differences between "old" and "new" EDASM are summarized in Appendix A.

"Old" EDASM consists of six different program modules. These modules are . . .

PARTS OF "OLD" EDASM
EDASM . . . . An Applesoft loader
INTEDASM . . . . An Integer Loader
ASMIDSTAMP . . . . Header Identifier
EDASM.OBJ0 . . . . . Supervisory Module
EDITOR . . . . Editing Module
ASSM . . . . Assembly Module

EDASM has two operating modes. You use the *editing* mode to enter, correct, or change a source code. You use the *assembly* mode to assemble a previously completed source code into working object code.

To save on memory space, "old" EDASM either edits or assembles, but it cannot do both at once. There is a short "main" program to EDASM that is called EDASM.OBJ0. This program is the supervisor that calls up the editor module, called EDITOR, or the assembly module, called ASSM.

An option in "new" EDASM does let you do IIe co-resident editing and assembly.

Once you get EDASM.OBJ0 into memory, it will automatically load the editor and put you into the edit mode. Should you decide to assemble, EDASM.OBJ0 will automatically overwrite the editor and load the assembler and go ahead with assembly. When assembly is complete, EDASM.OBJ0 will automatically reload the editor module.

There are three ways to get EDASM running . . .

THREE WAYS TO BOOT "OLD" EDASM
RUN EDASM from Applesoft.
RUN INTEDASM from Integer BASIC.
BRUN EDASM.OBJ0 from the monitor.

The only use of the modules called EDASM and INTEDASM is to get you started from either BASIC and to handle the upcoming ID stamp. The program called EDASM is *not* EDASM!

Normally, you will be in Applesloth when you begin, so you probably will boot EDASM with the RUN EDASM command. If you use RUN INTEDASM instead, you must be in Integer, or at least have a copy of the Integer code in the machine. You can also *rerun* the assembler from the monitor or machine language by doing a BRUN EDASM.OBJ, but this will not let you load or edit the upcoming ID stamp.

I like to dedicate disks exclusively for assembly use. To do this, you write a special MENU or HELLO program that prompts "HIT SPACEBAR FOR EDASM." A space then autoboots RUN EDASM,

which in turn loads the actual EDASM modules. Each of these customized diskettes should have all the needed parts of EDASM on it.

Anyway, the "real" EDASM is called EDASM.OBJ. This is a short machine language supervisor that automatically switches you between edit and assembly modes by getting the EDITOR and ASSM modules off the diskette as needed.

One of the first things the EDASM or INTEDASM loader does is show you the current *ID Stamp* . . .

```
ID STAMP—

A page header that automatically tells the
name of the programmer, the date, and
the version number of an assembler
listing.
```

The ID stamp on EDASM is called, of all things, ASMIDSTAMP. When it comes up on the screen, you edit it to hold the correct date, your initials, and the version number you want your next assembly to be.

You can have up to 17 characters in the ID stamp. The final two characters will automatically increment on each new assembly. Anything at all handy can go here, but the date and your initials is more or less standard.

Here's the ID stamp I use . . .

30-MAR-83 DEL #01

If you decide to activate this ID stamp by using the SBTL pseudo-op at the beginning of your program, it will automatically appear on the top right of each page of your assembly listing. At the same time, the name of your source code file will appear at the top left.

It is a good idea to always use the ID stamp.

The ProDOS version of "new" EDASM has a more restrictive ID stamp, but is compatible with a time and date clock card.

An error message gets generated if you do not have the ASMIDSTAMP file on your diskette or have locked it. This was a fatal error on "old" EDASM that stopped the assembly process. "New" EDASM has fixed this bug. We'll see more on error messages in chapter six.

You also must have at least two digits in your ID number, because otherwise a "#9" increments itself into a "$0," rather than a "10." This happens because the ASCII code for "$" is one more than the code for "#."

Actually, you can auto-increment version numbers up to 999999, or six places in the ID stamp. But you can safely use the "hundreds" slot for a space, a "#," or any other character if you are sure you will never increment past version #99.

Here's a summary of the ASMIDSTAMP rules . . .

ASMIDSTAMP RULES

ASMIDSTAMP must be on the diskette and must be unlocked.

ASMIDSTAMP is entered and edited only by the loaders called EDASM or INTEDASM.

The actual ID stamp is activated with a SBTL pseudo-op as the first line in a source code listing.

You are only allowed 17 ID characters.

Characters 12 through 17 are used for the version number and will auto-increment on each successive assembly, even if these characters are not ASCII numerals.

Normally, you edit the ID stamp only once at the beginning of a session. The ID stamp physically sits in Apple memory locations $02B8 to $02C8, equal to decimal 952 to 968. The ID stamp also gets rewritten to diskette every time it is incremented. Thus, you can rerun the assembler from the monitor with a BRUN EDASM.OBJ command if these locations still hold the running ID stamp.

Before we get into just how to use EDASM, here are a few general use hints . . .

It is usually easier to edit an existing source code than create a new one.

Editing of most source code should be done BACKWARD, working from finish to start.

NEVER overwrite disk files.

ALWAYS use a new version number for any revision or update.

While we aren't quite ready for details like these, we've put these rules up front since they are so important.

We'll show you later how to use the module called EMPTY SHELL.SOURCE as a "tree" to hang your source code on. Normally, you'll start either with the empty shell or some of your older source code, and then remove what you don't need and overwrite what you do. Your *initial* entry or editing of your source code should be done in the normal way, going from beginning to end.

But, any time you add, edit, remove, or change anything from a source code file, always work *backward* starting with the *highest* numbers. The reason for this is . . .

> **The reason you edit BACKWARD is so that the line numbers do not change as a result of the editing you have already done.**

Say you add a line in the middle of your program during editing. All successive line numbers will increase. On the other hand, if you delete a line, all successive line numbers will decrease. Either way means that all line numbers are wrong beyond where you now sit. Any reference to altered line numbers on an already *existing* printout or assembler listing will be wrong.

So, always edit from high numbers to low numbers.

We've already seen that you must have separate names for your source code and your object code, and that you should never over-write existing files. Always add a new and higher version number as you go along. This way, should you bomb the program or do something else really dumb, you can always fall back on your last working version.

If you run out of disk space, you can always go back and delete very old versions of your code. But, never delete the next-to-oldest version, or any "landmark" versions where something difficult just got completed.

OK, so how do we talk to EDASM and tell it what we want it to do? You communicate with EDASM with simple keystrokes that go by the fancy name of *editing commands . . .*

> **EDITING COMMANDS—**
>
> **Any instructions you send EDASM via the keyboard that are used to create or edit a source file, access a disk, control a printer, get tutorial help, or begin the assembly process.**

Editing commands are the way you tell the assembler to edit or assemble source code for you. There are dozens of these that we will shortly look at. We'll note in passing that the difference between an editing command and a pseudo-op is that an editing command does something right now, while the pseudo-op does something only when it crops up on a line during an assembly.

As a reminder, the word "editor" has at least two meanings when you are using an assembler. An editor here is a program module that lets you enter, modify, and save source code. The edit mode of an editor module specifically lets you make line-by-line changes to an existing source code. Thus, only some of the editing commands actually do line-by-line editing. Usually the context will help you out here.

If not, take a guess.

We can group editing commands several different ways. Here is how I would do it . . .

```
┌─────────────────────────────────────────┐
│                                         │
│        TYPES OF EDITING COMMANDS        │
│                                         │
├─────────────────────────────────────────┤
│                                         │
│   Disk . . . . . . . . Accesses disk files       │
│   Print . . . . . . . . Controls the printer     │
│   Entry . . . . . . . Creates source code        │
│   Edit . . . . . . . . Changes source code       │
│   Assembly. . . . Creates object code            │
│                                         │
└─────────────────────────────────────────┘
```

Disk editing commands let you access the diskette. Remember that while editing, you are only working with source code and that these source codes in EDASM are usually disk-based text files.

Print editing commands let you turn the printer on and off. But, remember that for most uses, you do not make a hard copy printout of your source code. Instead, you generate a combined assembler listing at the time of assembly. The assembler listing includes both source code and object code.

Entry editing commands are the housekeeping stuff needed to let you start a file, add to it, delete lines, do a listing, search or change, get tutorial help, and so on.

Edit editing commands do the actual changing on a single line of source code. On EDASM, editing is a dependent mode you switch in to and out of. Unfortunately, the "old way" edit editing mode is a separate world all its own.

Finally, the assembly commands take an existing source file and begin the assembly process. We will save the assembly commands for chapter six.

Let's look at these editing commands, more or less in order of importance, like we did earlier with the pseudo-ops. As before, we'll call anything that's not too often used or too important an "also ran" and not go into too much detail on them. Later on, if you need one of these for special effects, you can look them up on your own in the EDASM manual. See Appendix A for differences involving "new" EDASM.

DOS Editing Commands

We'll look at these first, since most everything EDASM does is involved one way or another with disk access.

LOAD

The LOAD command is used to get a source file off of the disk and into the RAM used by EDASM. LOAD automatically destroys everything in RAM and starts you over fresh. Note that this is a *text* file loader, unlike the usual DOS load commands.

The usual command is LOAD ZORCH.SOURCE 1.0, where ZORCH.SOURCE 1.0 is the textfile name and version of your previously existing source code. On "old" EDASM, you must specify the slot and drive *before* loading, and not with commas added to the load command.

Should you try to LOAD a nonexistent or misspelled file name, or one off the wrong drive, LOAD will create a "new" file with this name

that will return to haunt you. A surprisingly fast load is one hint that this is happening. When this happens, it pays to get rid of the "empty" textfile *immediately,* for it surely will cause you grief later.

A normal and correct load of a normal textfile will leave you with an OUT OF DATA error message. This is apparently considered kosher by "old" EDASM.

Here are some LOAD cautions . . .

LOAD HINTS

You normally load only TEXT files for source code use.

A wrong source code file name or slot or drive may create a new disk file having that wrong name.

An OUT OF DATA error message may happen normally with a good load and should be ignored.

All previous RAM contents usually get overwritten during a LOAD.

Slot and drive must be correctly set before a LOAD.

A load will return with a FILE TYPE MISMATCH error if you try to use a binary object code file. An I/O ERROR means you have a sick drive, a missing or uninitialized diskette, an open door, the wrong slot number, or something else equally bad. The ProDOS version of "new" EDASM does give you ways to use non-text files.

SAVE

The SAVE command takes the source code you have in memory and saves it under a file name. Usually, you will do a SAVE ZORCH.SOURCE 1.1, where ZORCH.SOURCE 1.1 is your text file save name and version number. As with the LOAD command, SAVE handles text files only.

You can just use the SAVE command and EDASM will save to the previous name, but this is bad for several reasons. The first is that you are overwriting code that at least did something, and the second is that you may have forgotten the previous name or changed it to something else.

It is also possible to save only certain lines to a file, using a command such as SAVE 36-78 SNORK.SOURCE 1.3. Naturally, you should pick names that are significantly different for modules that are only parts of programs or those that are utility or service subroutines from a library.

One extremely important rule is to SAVE before you assemble! Remember that EDASM is usually disk based. EDASM normally goes to the disk to get its source file during assembly. Your nice and neatly

edited file in RAM gets overwritten if you forget to SAVE it before doing any assembly . . .

SAVE HINTS

ALWAYS save source code to disk before starting any assembly.

NEVER overwrite a previous source code text file. ALWAYS use a higher version number.

Using SAVE without a filename is very bad practice.

ALWAYS use source code file names that are different from object code file names.

You can get all sorts of error messages on a SAVE. DISK FULL, WRITE PROTECTED and FILE LOCKED are obvious. I/O ERROR means the usual bad news. FILE TYPE MISMATCH means you have tried to save a text source code under the same name as a previously generated binary object code, or done something really dumb.

APPEND

The APPEND command lets you splice one source file onto the *end* of the existing source code file in RAM. Use LOAD to start over, and APPEND to add a new source code text file to the end of an existing one already in memory.

Unfortunately, "old" EDASM has no direct way to let you splice a utility or library module into the middle of existing source code. Instead, you go the long way around to APPEND the needed module onto the end of the existing source file. Then you use the upcoming COPY and DELETE commands to move the needed module to the middle of the code and then drop the extra code at the end.

Thus . . .

APPEND HINTS

The APPEND command tacks a source code file onto the end of another source code already in memory.

To insert something in the middle of existing source code, APPEND to the end, COPY to the middle, and DELete the duplication at the end.

You can also use line numbers with APPEND, but if you do, APPEND *overwrites,* rather than inserts, the existing source code. This

is dangerous unless you really want this sort of thing. See the EDASM manual for more details.

There seems to be some confusion between APPEND and CHAIN. APPEND is a way to immediately add one source code file to the end of a second source code file already in memory. CHAIN is a pseudo-op used to switch over to and continue the assembly process by switching to a second source code file still on disk. Use APPEND from the keyboard while editing. Use CHAIN from the source code while assembling.

You'll get the same types of DOS error messages with APPEND as you will with LOAD. If you try to overfill the available RAM space, you will get an OUT OF MEMORY error message.

"New" EDASM gives you several powerful "INCLUDE" and macro ways of combining "library" source code modules into a source code.

SLOT
DRIVE

These are used to change the slot and drive as needed. Typical commands are SLOT6 or DRIVE2. Default values are slot 6 and drive 1.

A gotcha . . .

> **On "old" EDASM, SLOT and DRIVE had to be properly set *before* doing a LOAD, APPEND, or SAVE. "New" EDASM eases this restriction.**

SLOT may be abbreviated to SL and DRIVE may be shortened to DR. Note that you can *not* change slot or drive in "old" EDASM with commas and trailing commands. Thus, LOAD SNARF.SOURCE,D2 was a no-no. Use DR2 <cr> LOAD SNARF.SOURCE instead.

Typically, you will use a single drive as slot 6, drive 1, and a pair of drives as slot 6, drive 1 and drive 2. All the EDASM modules should normally be in drive 1.

The EDASM modules are short enough that I like to keep a copy of them on all the source code and object code diskettes. Thus, slot and drive should rarely need changing.

CATALOG

This is like your usual disk CATALOG command, except it can be abbreviated to CAT. Obviously, you use it to find out what is on your disk, or to make sure you have the right diskette in the correct drive.

. DELETE
. LOCK
. RENAME
. UNLOCK

Note the periods that start these commands. For direct DOS access, you start the command with a period and then follow that period with a legal DOS command . . .

> **For direct access to DOS, use a period before the DOS command.**

We already have ways to do a catalog, load, save, append, and to change the slot and drive inside EDASM as EDASM editing commands. Anything else that involves DOS needs the period in front.

The most common direct DOS commands you would use are .DELETE, .LOCK, .RENAME, and .UNLOCK. These are used in the usual way with the usual rules. LOAD, SAVE, CATALOG, and APPEND are already available as unique EDASM commands, so you do not have to use periods on these.

In fact, nasty things can happen if you try a ".LOAD" or a ".SAVE." Don't use the period in front when it is not needed.

Note that the .DELETE command is used to delete a file on a diskette. There are different ways to remove a line from the source code or a character from a line. We will look at DEL and [D] shortly.

By the way . . .

> **It pays to keep ALL files on any EDASM disk locked at all times.**
>
> **The only exception is ASMIDSTAMP.**
>
> **You should immediately lock a new source file as soon as it hits the diskette.**

The reason for this is that one moment's careless mistyping or lack of coherence can eliminate a year's work. Keep *everything* locked at all times, except for the ASMIDSTAMP.

ALSO RANS—

There are three other disk edit commands provided in "old" EDASM. These are FILE, TLOAD, and TSAVE.

The FILE edit command gives you the name of the last file you loaded or saved. While this can be handy as a reminder or a memory jogger, remember that it is bad practice to *ever* save new source code to a previous file name.

The TLOAD and TSAVE file commands are intended to allow cassette tape loading and saving of source code. Such use of EDASM is both obsolete and insane.

The ProDOS version of "new" EDASM has a number of other disk commands. See Appendix A.

Print Editing Commands

The print editing commands let you decide whether your assembly listing is going to go to a printer or to the video screen.

Once again, this reminder . . .

> **You do not normally make hard-copy records of your EDASM source code.**
>
> **Instead, use an assembler listing that combines both source code and object code into one printout.**

The advantages of the assembler listing are that it shows you both source and object code together. Errors and mixups are also minimized, since you cannot create a legal assembly listing of code that will not assemble.

PR#0
PR#1

The PR#0 command tells the assembler that, *later on,* it is to assemble only to the video screen while it is routing the object code to the diskette.

The PR#1 command tells the assembler that it is *later* to make a printed record of the assembly listing while it is generating the object code for the diskette.

Nothing immediate happens on this PR#1 command, and the printer does *not* turn on until a later assembly. This is the normal and usual way of deciding whether you want a printed assembly listing or not. It is always a good idea to always do a printed assembly listing, even if it does take longer and is noisy.

If for some reason you really wanted to print your "pure" source code listings, or else wanted to "log" your assembly work session, you can use the .PR#1 command to immediately turn the printer on, and could use the .PR#0 command to return to the video screen. Note the periods on these commands, and note further that there is normally no good reason or excuse to be doing this.

At any rate, for normal use, you can follow the PR#1 command with a string you want to send your printer, such as PR#1, [I] 80N, or whatever. This can be used to set the printer font, line width, screen echo, or anything else you feel is needed.

It's easy to forget, so . . .

> **Always use the PR#1 command before an ASM command, or you may not get a hard copy of your assembler listing.**

An old PR#1 command gets remembered only so long as EDASM is in the machine and alive. If you turn the power off or exit EDASM, a new PR#1 will be needed every time you want a printed assembly listing.

Once again, remember that the PR#1 command does not instantly turn the printer on. It waits until the assembly process is to begin, and then activates the printer long enough to make one hard copy of the assembly listing.

The ProDOS version of "new" EDASM also gives you a "print to

disk" listing option. This can be most handy for "camera ready" print upgrades, typesetting, or telecommunications. The DOS 3.3 versions of EDASM lack this feature.

Entry Editing Commands

The editing commands used to enter source code into RAM are the ones you will use the most. We purposely have lead with the DOS commands since EDASM is a DOS-based assembler.

Let's now look at these workhorse entry commands in detail.

?(HELP)

A question mark gives you two screens worth of help in the form of an alphabetical listing of all the EDASM commands, their syntax, and their allowable abbreviations. Many of the EDASM commands can be shortened to one or two keystrokes to save time and hassle. If you ever are in doubt as to who does what, use this help screen to double check.

NEW

This command "erases" memory and lets you start over on a new source code listing. Note that the LOAD command also implies a NEW command, since loading destroys the old memory contents. Should disaster strike, "new" EDASM gives you a way to "undo" this command.

ADD

This command picks up at the end of the existing source code file and lets you add new lines to the end. If your source code has zero length, you start "adding" with line number one. To begin creating a new source code, you must first use NEW, followed by ADD.

INSERT

This command lets you shove stuff into the middle of an existing source code file. You must give the number of *the first line that is to be bumped below the new stuff.* Thus, an INSERT6 will let you start adding things after the existing line 5 but before the existing line 6. Line 5 stays line 5, but line 6 becomes lines 7, 8, 9, or whatever as you add entries.

[Q] (QUIT)

The [Q] command gets you out of either the ADD or the INSERT mode and returns you to the main editor program. (We will use the "WPL" method of showing control characters here. "[Q]" means "<CTRL>Q.")

This gets confusing fast, since there are four different ways you might like to exit something . . .

"TERMINATE" COMMANDS
[Q] quits you from ADD or INSERT.
[X] exits you out of EDIT.
[C] cancels a long LIST.
END ends your EDASM session.

This is kind of poor, but that's the way it is. It would be far better to have a consistent exit process for each part of each module, but "old" EDASM does not do this.

If you use the wrong exit command, you can force strange characters and control commands into your source code files that will cause all sorts of nasty hassles later. Empty line numbers, or else line numbers with funny characters or embedded control codes can result. Watch this detail closely.

[Q]uit entry. [X]it editing. [C]ease listing. END session.

END

As we've just seen, the END command is used when you are finished with EDASM. END totally exits the program. END usually returns you to Applesloth.

You might like to do this when you are finished for the day or else might like to try some object code out to see if it works.

Unlike some assemblers, EDASM does not let you test object code from within. To do a test, you give EDASM an END command, and then BRUN your object code . . .

You must exit EDASM with an END command any time you want to test your object code.

The good news about not being co-resident is that there is no way the object code can plow EDASM if EDASM is not in the machine or no longer in use. This means your object code can be tested and run exactly where it belongs in the Apple.

The bad news, of course, is that the round trip of edit-assemble-test gets to be a hassle.

This round trip time can be dramatically minimized, but not eliminated, by linking EDASM to newer and faster versions of DOS or disk emulators. Using LST OFF to drop the label reference dump helps bunches as well. "New" EDASM does give you limited co-resident and in-place assembly options.

Once your test is complete, you can use RUN EDASM or RUN INTEDASM as needed to get the assembler back into the machine. You alternately do a BRUN EDASM.OBJ if you are sure the ID stamp is still at $02D8. Remember to turn PR#1 back on each time you reenter EDASM, or you will not get any hard copy.

LIST

This is the command to list all or part of your program to the screen. There are several ways to use LIST . . .

USING LIST	
L	—Lists everything
L10	—Lists line 10
L10,20	—Lists line 10, then 20
L20,10	—Lists line 20, then 10
L10-20	—Lists lines 10 thru 20
L10-	—Lists lines from 10 on
Spacebar	—Single steps listing
[C]	—Cancels listing

As we've shown, LIST can be shortened to L. You probably will use this edit command more than any other, for it shows you what you have in your source code and where it is located. "New" EDASM gives you additional LIST options. See Appendix A.

As as the chart shows us, you can stop a long listing by hitting the spacebar. Each new hit of the spacebar gives you one more line sent to the video screen. You can cancel a listing with a [C].

Should you get nothing on a LIST command, either you have an empty source code or else you told it to list backward or did something else weird.

DELETE

This is a dangerous one. DELETE, or simply D will eliminate one or more lines from your source code, and will automatically and *immediately* change the numbering of all lines above the ones you deleted.

Here are some legal ways of . . .

USING DELETE	
(smart and safe)	
D6	—Deletes old line 6
D6-10	—Deletes old lines 6 to 10
D6-999	—Deletes all lines above 5
D8,7,6	—Deletes old lines 8, 7, 6
(dumb and deadly)	
D6,7,8	—Deletes old lines 6, 8, 10
D6,6,6	—Deletes old lines 6, 7, 8

What you have to remember is that each deletion immediately bumps all line numbers above the deletion down by one. So, it is safe

to delete a single line. It is safe to delete a range of lines. It is safe to delete everything above a certain line.

It is also safe to delete any number of widely spaced individual lines *in reverse order*. But it is deadly to delete individual lines in "front-ward" order, since the line numbers will change between the time you call for the deletion and the time the deletion is actually done.

In the examples above, a D8, 7, 6 command works OK, since the deletion of eight bumps nine down to eight but doesn't hurt seven or six. But a D6, 7, 8, deletes six, converting line eight to line seven. It then deletes the old line eight, which is now the new line seven. Finally, line ten gets deleted since the previous two deletions has bumped ten down to eight.

Similarly, a D6, 6, 6, is one very bizarre way to delete lines six, seven, and eight of an existing source code.

The rules are . . .

> **ALWAYS delete individual lines in reverse order, using the HIGHEST line number first!**
>
> **ALWAYS relist after any deletion since the line numbers will have changed.**

A D1-999 command is the same as NEW as it eliminates the source code file entirely. You can use any number higher than your *highest* line number. I usually use 999.

Note that this particular DELETE command dumps whole source code lines only. Use the .DELETE command instead when you want to erase disk files. There is yet another way used to delete single characters inside a source code line, and involving [D]. More on this shortly.

COPY

The COPY entry editing command is one good way to duplicate a group of lines. This can be most handy when you have many lines that differ slightly. It is usually far easier to copy and then edit than it is to enter everything twice by hand. Working files are one very useful place to be COPYing. So is any code that gets repeated "alike but different somehow."

You have to tell EDASM how much you want copied where. For instance, COPY 20-30 TO 53 will put an image of the eleven lines from 20 through 30 so that a new image starts with line 53. The old line 53 will now be line 64.

There is no MOVE command as such in "old" EDASM. Instead, you COPY and then DELETE. In the previous example, you might delete lines 20 through 30 if you were only interested in a move. "New" EDASM does give you a REPLACE option.

We've already seen that you cannot load a disk module into the middle of your source code. Instead, you first APPEND the new stuff to the end of your source code, then COPY it to where you really want it, and then, after checking line numbers very carefully, delete the extra stuff at the end.

Sort of roundabout, but it works. "New way" editing eliminates these hassles completely.

LENGTH

The LENGTH entry command, which can be shortened to LEN, tells you how long your existing source code is, and how much RAM you have remaining in the Apple for additional lines.

Typically . . .

> **EDASM holds a source code file of roughly 29K characters in its RAM.**
>
> **This is enough for 700 to 1400 lines of typical source code.**

Thus, there is more than enough room to put fairly long source codes into EDASM all at once. If you need more room than this, there is always the CHAIN pseudo-op used during assembly that can tie any number of source code files together for virtually any length assembly.

Should you try to overfill RAM, you will get an ERR: MEMORY FULL message. Usually this is not destructive. All it means is that it is time to "garbage collect" and eliminate any source code lines you aren't using anymore. Otherwise, you can split the source code into two or more pieces.

Source code listings tend to use far fewer characters than do word processing files, and EDASM has more source code space than some word processors have file space. So, you aren't likely to run out of source code room for all but the most humongous projects.

Don't skimp on documentation to try to save source code space. It is *never* worth it . . .

> **There is more than enough room in EDASM to hold and edit a long and very well documented source code.**
>
> **NEVER skimp on documentation to try to save on memory space!**

The one big advantage of a disk-based assembler over an in-place assembler is that it lets you work on longer files with better documentation, since no room has to be set aside for object code inside RAM.

ALSO RANS—

There are eight other "old" EDASM entry edit commands that are specialized and do not see much use. These are PRINT, WHERE, TRUNCON, TRUNCOFF, LOMEM:, HIMEM:, MON, and TABS.

The PRINT command gives you a list without adding line numbers. This can be handy when you are editing something other than source code with EDASM, such as a BASIC program or some other text file.

Note that PRINT in this case has nothing whatsoever to do with hard copy. PRINT, like LIST, normally goes only to the screen. If you do really want to get a hard copy without line numbers, you can use a .PR#1 before the PRINT command, and a .PR#0 following. Note those periods. Such use is very rare, but you might find it handy for specialized editing.

The WHERE command returns the hexadecimal address of a source code line number. For instance, WHERE14 will give you the starting address of source code line number 14.

Finding the actual location of a line in memory is sometimes useful if you want to directly modify source code from the monitor. This is one very inelegant way to handle lowercase alphabets under "old" EDASM on an "old" Apple. WHERE can also be used to find hidden control characters and to resolve other problems.

A source code line can be anything, but most programmers end up with something more than 40 and less than 80 characters on most of their work. This means that a listed line will not fit on a single line of an older Apple II 40-column text display.

So, when you are listing, the comments are usually dropped and do not appear on the 40-column Apple screen. To be able to see everything on screen, at the cost of having the comments interspersed with the op codes and operands, you can use the TRUNCON or simply TRON command. To get back to the usual display, use TRUNCOFF or TROF.

Truncating only affects the display during a list. It does not change anything in the machine and is not present during editing.

"New" EDASM on a IIe does, of course, give you a choice of 40- or 80-column display.

The HIMEM: and LOMEM: commands may be used to change the available space for EDASM source code lines. You might do this if you want to try to assemble and debug at the same time, or if you want to protect some memory space for some other reason.

The default values are LOMEM:8192 and HIMEM:38400 on "old" EDASM. Which means your source code file space goes from hex $2000 to $9600. You can normally *raise* LOMEM or else *lower* HIMEM to shorten your source code workspace. If you try going the other way on a stock Apple, you will end up plowing EDASM on the low end and overwriting DOS buffers on the high end.

One interesting possibility is to move DOS to a RAM card. You could then set HIMEM to 49151 and pick up another 10751 bytes of RAM for your source code. Which would gain you several hundred more source code lines.

The MON command is particularly dangerous in inexperienced hands. This entry edit command moves you directly into the monitor from EDASM for special uses. Any of the damage you can normally do from the monitor is now available for you with which to ruin your source code. A 3D0G gets you back to EDASM if anything is left.

The editing portion of EDASM was initially set up to edit just about anything, and not just 6502 op codes for the Apple II or IIe. Remember that "old" EDASM uses the spacebar as a tab to move between source code fields. For special uses, you might like to tab on a different character, tab to different positions, or else not tab at all.

To handle this, there is a TAB command. The usual format is TAB6,13,25,37,Z. This particular command says we are to tab anytime the character Z shows up, going to the next available horizontal position in the list. Using TAB without a list totally defeats any tabbing. Tabs are limited to 40 or fewer horizontal positions in "old" EDASM.

Normally, you wouldn't mess with the TAB commands. One place different tab settings might be handy is if you are cross assembling

code for some machine using a non-6502 microprocessor, and wanted different width columns in different places.

Sometimes you will want to turn all tabbing off to simplify editing comments. More on this shortly.

Edit Editing Commands

So far, all our entry commands did things to entire source code lines. It is also nice to be able to modify individual characters and individual parts of a single line. EDASM has a special editing mode that lets you do this.

Unfortunately, the editing is mode dependent, which means that you can't edit anything directly on screen or from a list. Instead, you turn the editor on, edit the line, and then go back to the entry mode.

Dumb. Dino even.

As a reminder, "new way" editing can eliminate this weirdness in one swell foop.

EDIT

The EDASM command to edit one or more lines is called, strangely enough, EDIT and may be shortened to E. What EDIT does is bring one line at a time up on screen where you can mess with the line to your heart's content. After editing, you accept the line, and the EDIT command then goes on to any further commands you have given it.

Here are a few examples . . .

USING EDIT
E6   —Edits line 6 only
E6-10 —Edits lines 6 thru 10
E6-   —Edits lines 6 thru all
E-   —Edits everything
E9,7,3 —Edits 9, then 7, then 3
E3,7,9 —Edits 3, then 7, then 9

You can edit any number of lines in any order, but if you are reediting a source code and are making changes that involve deletions and insertions as well as editing, you should *always* start from the high numbers and work your way down.

Any of the edit commands puts a line on screen so you can edit it. Once you have the line on screen, you can then use additional special edit editing commands to make any changes.

Here's a summary . . .

EDIT EDITING COMMANDS	
→	—Move one to right
←	—Move one to left
[return]	—Accept line as is
[T]	—Truncate and accept
[R]	—Restore line as it was
[D]	—Delete character
[I]	—Insert character
[F]	—Find character
[V]	—Verbatim entry
[X]	—End edit editing mode

Let's look at these edit editing commands one at a time.

Once again, this is a line-based editor, so you work with the edit editing commands to change one source code line at a time. When you are happy with one line, you then go on to another line.

As a reminder, we will use "WPL" notation in the text that follows. Thus, "[T]" means to press and hold the <CTRL> key. Then press and release the "T" key. Then release the <CTRL> key.

Starting with the obvious, to get the line on screen, you give an EDIT or E command that brings up the line number you want. You'll see your line and a winking cursor. The right arrow moves you one character to the right and the left arrow backs you up one, just as you would expect.

If you want to change the character the cursor is sitting on, just overtype it. After a change, the cursor moves one to the right to let you continue changing characters in the order you would expect. This is just like using a word processor in its "replace" mode, except you are working only on a single line at a time.

Putting last things first, once you get the line edited the way you really want it, a carriage return accepts the line and enters it back into the EDASM source code. So, use a carriage return to accept the line as you now see it.

Sometimes your editing process will leave a bunch of characters or unwanted garbage hanging on the end of the line you really want. In this case, you use a [T] to *truncate* and accept the line. This is somewhat similar to a word processor command of "Erase to End of Line."

Thus, a carriage return accepts the entire line. A [T] accepts only the part of the line that is to the left of the cursor. The cursed character is dropped, as is everything else to the right.

It is real easy to make a royal mess of the line you are working on, plowing it up beyond hope. If this happens, just type [R] to *replace* the original line you first were editing on screen. All the damage miraculously vanishes, and you can try again.

To delete a character, just put the cursor to the left of it and hit [D]. Each successive hitting of [D] swallows one more character and shortens the line by one character. Typing anything else, such as a right or a left arrow, exits you from this deletion sub-mode.

To insert one or more characters, again put the cursor just to the left of where you want the insertion to take place. Then type [I]. This turns on the insertion sub-mode. All characters you now type will be added, one at a time, to the line you are editing. The line, of course, will get longer as you do this.

To exit the insertion mode, use a right arrow, a left arrow, a carriage return, a [T], or an [R]. The arrows leave you editing the same line. The carriage return accepts that line as is. Using [T] truncates and accepts only the stuff to the left of the cursor. Hitting [R] aborts what you have on screen and gives you the original line back.

There are two "also ran" editing commands. These are [V] and [F].

The [V] command is dangerous and tricky. The V here stands for *verbatim,* and this is how you can force control commands into your source code line. Verbatim entry might be needed to imbed special printer commands or to do other very special things with your source code lines.

A second [V] exits the verbatim mode. Use of [V] is very dangerous and should be avoided unless you really know what you are up to.

The [F] command *finds* the next occurrence of a chosen character in the line. For instance, "[F]H" moves the cursor to the first capital H it finds on the source code line. In this example, the control key is only held down for the "F," and *not* the "H." If it doesn't find a capital H on the line, the cursor stays where it is. Thus [F] is sort of an "express" right arrow.

Note that [F] is a "cursor positioner" only. There is also a second and much more powerful way to find and replace things. More on this shortly.

If you decide you do not want to edit anymore, just use [X] to "X-it" the edit mode. This might happen if you decide to leave the original line the way it was after all. You might also want to do an [X] if you told the editor to give you bunches of lines to edit, and you got past the lines you really were interested in.

Once again, note that the "stop" commands are inconsistent in EDASM. A [Q] quits entry or insertion. A [C] cancels a listing. Use of [X] exits the edit editing mode, and finally, END gets you completely out of EDASM.

EDASM's editing commands are fairly powerful as far as line-oriented editor commands go. A little practice and you will be able to edit with a vengence. But, watch what you are doing. If you try to exit the editing mode with a [Q], or try stopping the entry mode with an [X], you will end up in deep trouble.

To summarize, use E or EDIT to bring one line at a time on-screen for editing. Position with the right and left arrows, or use [F] as an express right arrow. Make corrections at the cursor by overtyping. Use [D] to delete and [I] to insert. To accept the whole line, use a carriage return. To accept everything up to the cursor, use [T]. If you want to reject the line and start editing it over, use [R]. To exit the edit editing mode, use [X].

FIND

Besides there being commands to edit single EDASM source code lines, you can also do "global" search and replace on the whole

source code. Three commands involved here are FIND, CHANGE, and REPLACE.

A command of FIND"$8D" will find and list all lines on which the string $8D appears. The quotes here are delimiters and are not part of the string. This is handy to locate a line if you messed up the line numbers. You can also search only part of the source code by putting line numbers between FIND and the search string. For instance, FIND6-30"SNORK" will search lines 6 through 30 for the label SNORK.

An [A] may be used as a wildcard in your find. Thus FIND "[A]HIS" will search for all four-letter sequences that start with any character but end with HIS. This is sort of specialized. One use is to find two different ranges of addresses, such as might happen on HIRES1 and HIRES2 graphics. Another use is to find a word whose first letter may or may not be capitalized.

CHANGE

The real heavy search and replace is called CHANGE. To use CHANGE for a global search and replace, use three delimiters separating the "old" and "new" string.

For instance, a command of CHANGE/ZILCH/ZORCH/ will first ask you "ALL OR SOME?" If you recklessly answer ALL, the command will go through your entire source code and change every ZORCH it finds to ZILCH.

If you more sanely answer SOME, the command will go to the first ZORCH it finds and show you the change on the first line. If you want the change made, use [C] to accept the change. Use [ESC] to abort the search and use any other key to continue searching.

As with FIND, you can use a range of line numbers and you can use the [A] wildcard in the original string.

A caution . . .

> **Do not use CHANGE ALL.**
>
> **Use CHANGE SOME instead.**
>
> **It is too easy to have a character string buried somewhere else, such as in a label or comment. ALL can replace more than you expect, in all of the wrong places.**

It is too easy to make some change and have that same change crop up inside labels and within comments.

For instance, suppose you decide to change the state of a carry flag with a CHANGE/SEC/CLC, and then respond with an A for ALL. Sure enough, all the SEC op codes magically change to CLC. But a comment that had the upper case word SECONDS in it mysteriously changes to CLCONDS as well. No matter how "safe" an automatic change seems, it will always find some label or some comment *somewhere* to foul up the works.

There is one final class of editing commands that do the actual assembly for us. We will hold on these until chapter six.

AN EDITING HINT

One of the nastiest "features" of EDASM is that it tabs *everything*. Which is fine for real source code lines. But, any comment lines you use will have great heaping holes in them when you try to "old way" edit them or list them to the screen.

This means it is normally just about impossible to edit a nice title block and insert or change comments and still have them end up where you want.

There is a sneaky way around this . . .

TO EDIT COMMENTS—

Turn all EDASM tab settings off by using a single "T" command.

The comments will now all be in one piece and appear as they will in your assembly listing.

When finished editing the comments, reset the "old" EDASM tabs back to normal with a "T14,19,29" command.

Rebooting EDASM will also reset the tabs back to normal.

I hate to admit this, but it took me years to find this hidden trick. Try editing your comments both ways; you'll immediately see how powerful this stunt is.

Similar default tab values for "new" EDASM are "T16,22,36."

A LABEL LIST

One problem that crops up when you start writing your source code is that you may define a label and, a few lines later, that label will scroll off screen. You then forget what the label was or else misspell it, creating all sorts of problems.

Now, once you have your source code put together far enough that it can be assembled, the references at the end of your assembly listing can be turned on to give you a list of all labels. This is handy once or twice, but gets tedious if done each and every assembly.

Now, the dino people would have you pre-define or pre-equate each label before you actually start programming. This is a bad scene, though, since often the process of entering source code suggests newer and better ways of attacking your problem. So, it is best to stay flexible.

But how do you keep track of labels?

Here's a form that shows you one way . . .

```
┌─────────────────────────────────────────────────┐
│  LABEL LIST FOR [                             ]   │
│                                                   │
│  DONE BY [          ]   ASSEMBLER [            ]  │
│  DATE     [  -  -  ]     SYSTEM  [            ]   │
│  VERSION [        ]               [            ]  │
│                                                   │
```

LABEL	EQU	LINE	DFB	VALUE	USE

```
│  NOTES _____   PAGE [] OF []  │
│        _____         [      ] │
│        _____                  │
└─────────────────────────────────────────────────┘
```

What you do is go along and fill in the list as you define your labels. You also note whether the label is an EQU that is defined *before* the source code, a line pointer that is defined *inside* the working source code, or a DFB that is defined in files that usually *follow* the source code.

Here's an example . . .

LABEL	EQU	LINE	DFB	VALUE	USE

LABEL LIST FOR OBNOXIOUS SOUNDS

DONE BY DEL   ASSEMBLER EDASM

DATE 6-16-83   SYSTEM APPLE II/IIE

VERSION 1.0   FOR SAMS ASSY COOKBOOK

LABEL	EQU	LINE	DFB	VALUE	USE
HOME	✓			$FCA8	MONITOR SUB TO CLEAR SCREEN
INIT	✓			$FB2F	SUB TO SET UP TEXT SCREEN
SPKR	✓			$C030	SPEAKER CLICK LOCATION
WAIT	✓			$FCA8	MONITOR TIME DELAY SUB
DEMO4		✓		$7800	START OF DEMO
NXTNOT4		✓		$7809	LOOP TO PLAY NEXT SOUND
STALL4		✓		$780F	BETWEEN SOUND DELAY
DONE4		✓		$7821	DEMO EXIT
BASENT4		✓		$7822	BASIC ENTRY POINT
OBNOX4		✓		$7824	ML ENTRY POINT
LOK4		✓		$7830	LENGTH OK TO CONTINUE
SWEEP4		✓		$783F	SWEEP INITIAL SETUP
NXTSWP4		✓		$7841	SWEEP LOOP
NXTCYC4		✓		$7843	CYCLE LOOP
EXIT4		✓		$7859	OBNOX EXIT
TRPCNT4			✓	$785E	TRIP COUNTER
FLNGTH4			✓	$785F	# OF AVAILABLE SOUNDS
SEF0			✓	$7860	SOUND FILE START (TICK)
SEF1			✓	$7862	(WHOPIDOOP)
SEF2			✓	$7864	(PIP)
SEF3			✓	$7866	(PHASOR)
SEF4			✓	$7868	(MUSICAL SCALE)
SEF5			✓	$786A	(SHORT BRASS)
SEF6			✓	$786C	(MEDIUM BRASS)
SEF7			✓	$786E	(LONG BRASS)
SEF8			✓	$7870	(GEIGER)
SEF9			✓	$7872	(GLEEP)
SEF10			✓	$7874	(GLISSADE)
SEF11			✓	$7876	(QWIP)
SEF12			✓	$7878	(OBOE)
SEF13			✓	$787A	(FRENCH HORN)
SEF14			✓	$787C	(ENGLISH HORN)
SEF15			✓	$787E	(TIME BOMB)

NOTES   BASIC POKES SOUND TO
BASENT4 + 1   PAGE ☐ OF ☐

RIPOFF MOD. #4

We've put two copies of the blank label list into Appendix C. You can cut one of these out and blow it up to a better size on an enlarging copy machine.

The label list is most useful as a beginner's tool when you are first starting source code programming. After you have done an early assembly or two, the symbol lister will pretty much replace the need for the label list. So, this form is more to get you started than anything else.

Label lists and cross references are most handy, particularly if you are new to assembly language programming or are creating a totally new and different source code.

With "new way" editing, the split screen feature pretty much eliminates any need for a hand written label list.

NOW WHAT?

OK. You've gone through this chapter and the colon is *still* glowering at you, and everything here seems like a garbled mess. How do you make heads or tails out of anything?

How, in short, do you start writing source code?

Well, the first thing to do is to reread this chapter, and then reread the EDASM manual. We have purposely arranged things differently here, so you have two different ways that all of the EDASM commands get explained.

The second through ninety-ninth things to do are get some hands-on practice using and working with EDASM. And, after you gain practice, go back and reread this chapter and the parts of the EDASM manual that can be of help.

You'll have the best luck if you use the EDASM manual like you would an encyclopedia, going to it for a specific explanation of how a specific command works, rather than reading it from cover to cover. You can dramatically improve the EDASM manual by using page highlighters to emphasize what is important.

Chances are you will want to go on to the "new way" of entering and editing source code of the next chapter, since it is far simpler and much more fun. But, be sure to do some stuff the "old way" first. It is absolutely essential.

So, you boot EDASM. You customize the IDSTAMP. And, there's that colon. Now what?

OK. Do this.

Boot EDASM and make up a suitable ID Stamp. After the colon shows up, enter . . .

```
NEW <cr>
ADD <cr>
<sp> SBTL <cr>
<sp> SKP2 <cr>
<sp> ORG $0300 ; CODE AT $0300 <cr>
<sp> SKP2 <cr>
;    ***********************  <cr>
; *                        *  <cr>
; * THIS IS MY VERY FIRST  *  <cr>
; *      TRY AT EDASM       *  <cr>
; *                        *  <cr>
;    ***********************  <cr>
<sp> SKP3 <cr>
;          *** HOOKS *** <cr>
<sp> SKP1 <cr>
SPEAK EQU $C030 ; WHAP SPEAKER <cr>
WAIT EQU $FCA8 ; MONITOR TIME DELAY <cr>
<sp> SKP3 <cr>
;          *** MAIN PROGRAM *** <cr>
<sp> SKP1 <cr>
BLATT LDA #$22 ; SET TONE VALUE <cr>
<sp> JSR WAIT ; AND DELAY <cr>
<sp> BIT SPEAK ; WHAP SPEAKER <cr>
<sp> JMP BLATT ; AND REPEAT <cr>
<sp> PAGE <cr>
<ctrl>Q <cr>
SAVE MYFIRST.SOURCE <cr>
```

Just in case you are the one that never can find the key *marked* "ANY" on your Apple, the <sp> means "hit the spacebar once," and <cr> means "press the RETURN key once."

Here is what your on-screen source code should look like, following a LIST command . . .

```
 1            SBTL
 2            SKP2
 3            ORG  $0300        ; CODE AT 40300
 4            SKP2
 5  ; ************************
 6  ; *                      *
 7  ; * THIS IS MY VERY FIRST *
 8  ; *      TRY AT EDASM     *
 9  ; *                      *
10  ; ************************
11            SKP3
12  ;    *** HOOKS ***
13  ;        SKP1
14  SPEAK     EQU  $C030        ; WHAP SPEAKER
15  WAIT      EQU  $FCA8        ; MONITOR TIME DELAY
16            SKP3
17  ;    *** MAIN PROGRAM ***
18            SKP1
19  BLATT     LDA  #$22         ; SET TONE VALUE
20            JSR  WAIT         ; AND DELAY
21            BIT  SPEAK        ; WHAP SPEAKER
22            JMP  BLATT        ; AND REPEAT
23            PAGE
```

What should happen is that you should have generated a simple source code and then saved it to your diskette. You can then assemble and test the source code per the details in chapter six. Note that <sp> means "hit the spacebar" and that <cr> means "press carriage return."

The comment box will most likely be offset or broken up on your listing. This is caused by EDASM's tabbing all lines, regardless of whether they are comment lines or real source code lines. To get a box like the previous one, type "T," followed by "L." This will fix the comments but will mess up the real op code lines.

To put things back the way they belong on "old" EDASM, type "T14,19,29," followed by "L." This trick keeps you from going up the wall whenever you try to edit comment lines. On "new" EDASM, use "T16,22,36."

We'll look at the assembler listing you will get from this example later on in chapter six.

This program does do something. Try to analyze what it does and then, after assembling and testing your object code, try to change what it does.

Then, change your source code so it does what it is doing "for a while," rather than "forever." After that, make it do several different "for a whiles" in time sequence before stopping. Then rearrange that sequence into something a three year old can recognize.

The next step after you have written and assembled your own source code is to go back and reread the parts in the EDASM book, encyclopedia style, that apply to the commands you have used.

To keep the source code simple for your first try, we have simplified the usual stars, title blocks, spaces, pretty printing, and have omitted

the documentation and explanations that all decent source code *must* have. Besides, we are intentionally making a minor mystery out of what the code does. But the instant you get this program working, go back and put in *all* the documentation and extra pretty printing that you can think of.

What next?

From here, you can go into a "study" mode, or a "create" mode. To go the study route, load a ripoff module of your choice and modify it to make it do different things different ways. As you use new commands, go back and reread this chapter and the EDASM sections.

To go into the "create" mode, load the EMPTY SHELL.SOURCE ripoff module and then edit and modify this shell to properly document and contain your first program.

After that, you should be off and running. Once again, hands on is everything. What reads like so much gibberish becomes trivial *once* you actually go ahead and use the commands. Continually go back and reread.

The first few attempts at assembly programming are always frustrating, particularly if you don't pay attention to detail, or if you get upset or emotionally involved with seventeen error messages on a six-line program.

But it all will fall into place. What seems like superhuman effort and mind-boggling detail now will become trivial and of obvious second nature to you later.

Now this, this here's a football. See that big "H" over there?

MORE DIFFERENCES BETWEEN MY ASSEMBLER AND EDASM:

5

WRITING AND EDITING
SOURCE CODE
(the NEW way)

Word processors have come much further much faster than have the editors in assembly packages. The reasons for this are obvious, since anybody at all may want to process words, while only those very few of you that genuinely wish to become filthy rich will be using editors and assemblers.

Today, it turns out that . . .

> **A word processor usually makes a better editor than an editor does.**

There are compelling and powerful reasons to do your source code editing with a modern word processor, rather than using the editor that came with the assembler in the first place. The bottom line is that entry and editing get done a lot faster, have far fewer errors, and, best of all, are far more fun. So, that's what our "new way" of editing and entering source code is all about. You simply use the word processor of your choice instead of the editor that comes with your assembler.

Here's a long list of potential advantages of "new way" editing . . .

**"NEW WAY" EDITING
ADVANTAGES**

Free form entry
Modeless
Unbroken comments
Fewer round trips

Little new to learn
Lowercase and fancy art
Insert anywhere
Easy line and block copy

Glossary or custom keys
Renumbering on demand
Better disk access
Split screens

Status line
Fewer bad lines in code
Powerful search and replace
Supervisory macros

By "free form" editing, you have as much of your source code on screen at once as you want. You thus are able to continuously enter and edit. Being modeless means that there is no difference between entry, editing, and sub-editing activities, since you can do any of these at any time.

Many editors, including the older version of EDASM, tend to mess up comment lines with unwanted and unneeded tabs when you try to edit. With a "new way" word processor, you see the comments exactly as they will appear on the final assembly listing. It is now a trivial matter to insert another line inside a title block box, for "what you see is what you get," at least on comment lines.

There are fewer round trips through the edit-assemble-test process, since your comments and formatting are clearer, and there's much less of the fumble-fingeredness that gave you funny lines the old way. Besides, you tend to do more at once when it is easier. The "little new to learn" advantage assumes you already have and are using a word processor for everything else. You can ignore many of the tedious and specialized "old way" editing rules.

Lowercase comments and fancy border art are now trivial. It is also far easier to insert anything anywhere, and copy anything to anywhere, either whole blocks or individual lines. By using any glossary or custom keys on your word processor, you can gain bunches on speed, and you can now include *source code* macros, as well as the more common object code macros.

That hassle of the line numbers not matching an old assembly listing and causing problems can be eliminated entirely. Done the new way, you can turn your line numbers on and off at will. Better yet, you can *delay* renumbering until you are done editing and correcting. Thus,

line number 153 can *stay* line 153 even after you have added or removed lines lower than this.

Your disk access is now much simpler, since you can use trailing commas rather than the awkward SLOT and DRIVE commands that "old" EDASM demanded. The split-screen feature of a word processor is one of those things that may not seem important, but once you use it, you will never go back. Screen splits are particularly useful for finding label names elsewhere in the listing before it scrolls offscreen. With a split display, you can have, say, all the EQUs on one half, and your present entry point on the other half.

Other word processing goodies include a status line to tell you what you are up to, and much more powerful search and replace features. You also may be able to have longer source code files as well. This depends on your choice of word processor and assembler, but a large increase in source code file length is not all that unusual.

Our last advantage is a real biggy, but is available on very few (would you believe one?) word processors. If your word processor has a supervisory controlling language, you can program and automate many operations involved with writing and editing source code. This becomes the ultimate macro. We are, of course, talking about WPL, the supervisory language that goes with *Apple Writer IIe*. With WPL, things like numbering, delayed renumbering, and unnumbering become trivially easy, as does creating conditional source codes. And, once you get into a programmable word processor language, you'll be surprised at how much can be done how well.

Well, that's a pretty long list. But, what about the dark side? . . .

"NEW WAY" EDITING LIMITATIONS

The files may not be compatible.
Round-trip times will be longer.
Tabbing might be awkward.
Supervisory macros may not exist.

There are four main disadvantages to "new way" editing with a word processor. The first is that the word processor's files must be compatible with your assembler's source code files. Thus, not every word processor will work with every assembler package. If there are hassles involved in file format changes, then the "new way" may not be worth the effort.

Secondly, the edit-assemble-test round trip time will get pretty bad, since you have to separately load your word processor *and* your assembler and your final programs to be tested.

There are several ways to ease this particular hassle. Foremost is that new way editing usually saves you on the total *number* of round trips needed, since you are less likely to mess up comments or create fumble-finger errors, and since it is easier to do more at once. If you have an Apple IIe, with extended memory you can, in theory, patch things up so that your word processor, your assembler, *and* your object code can all sit co-resident in the machine. Just flip back and forth between main and auxiliary memory as needed. *There's also the*

obvious route of using a second Apple, which, these days isn't that far fetched.

Finally, you will need some way to number and renumber your source code lines. You may also want some way to do fancy tabbing. The easiest way to handle this is with a supervisory language that controls your word processor. A fully "open" word processor package that lets your own custom machine language codes directly interact with the RAM text file can also be used. This is fastest and best, but takes lots of effort. What all this means is that not every word processor is suitable for "new way" editing.

All in all, if you pick the right word processor, you will find the "new way" method far easier, far faster, far less error prone, and, above all, far more fun than the old way ever was.

The combination I use links EDASM with *Apple Writer IIe* with the WPL supervisory language. The results boggle the mind.

In fact . . .

> **The EDASM—Apple Writer IIe—WPL mix instantly converts one of the dreariest and dumpiest editor modules into one of the finest you can get, no holds barred.**

By the way, older versions of *Apple Writer* aren't nearly as good. *Apple Writer I* used noncompatible files. *Apple Writer II* will work OK, if your Apple has lowercase and 80 columns. If it lacks either of these, you won't gain nearly as much nearly as fast.

Linking these three packages is totally trivial. Just switch between them as needed. That's all there is to it! Before we look at some specific details, though, let's check into . . .
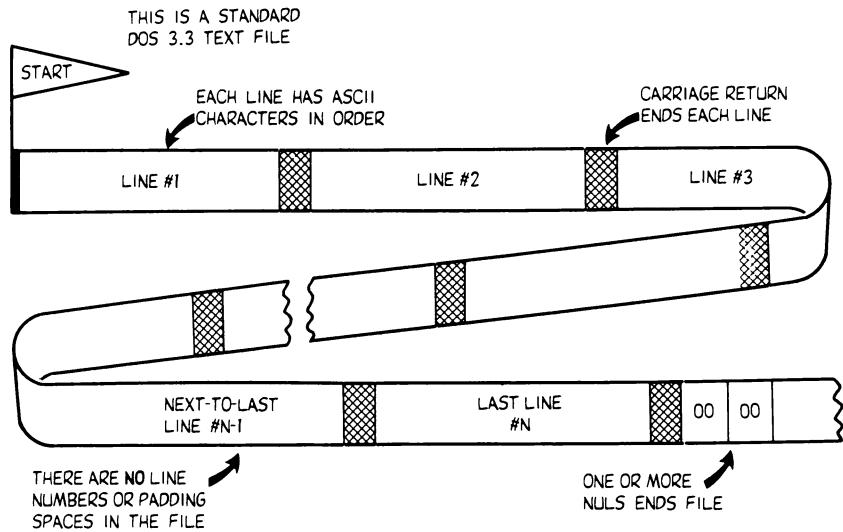
SOURCE CODE FILE STRUCTURE

If you are going to share a source code file between two programs, that file obviously has to be compatible and readable by both. Since EDASM uses standard text files, and since *Apple Writer IIe* uses standard text files, we shouldn't expect too many hassles. But you do have to know what goes where in the file.

So, your first task in starting "new way" editing is to find out all you can about what the files look like.

Here's what an EDASM source code file looks like when it goes to disk . . .

HOW EDASM SOURCE CODE IS STORED ON DISK:

Just as with any standard DOS text file, there are no header bytes that tell us an address or length. Each source code line consists of a string of ASCII characters that ends with a carriage return. The lines go onto the disk in sequential order, with line 1 first and the final line last. One or more double-zero NUL commands follow the final carriage return to mark the end of the file.

Two key points . . .

> **EDASM source code files do NOT have any line numbers in them.**
>
> **EDASM source code files do NOT have any "padded" tabbing spaces in them.**

There's not much point in filling up files with things that are unneeded or easily calculated. So, to save on source code length, there are no line numbers in the EDASM text files that hold your source code. The editor and assembler modules generate the line numbers for you when and as needed. They do this by counting the carriage returns as the lines come off the disk.

Similarly, on "old" EDASM there are no tab commands nor are there any "padded" extra spaces that fake a tab. The first three spaces are interpreted as tabs by the editor or assembler on any active lines.

By the way, we'll define . . .

> **Active Line—**
>
> **Source code line with an op code or pseudo-op in it.**
>
> **Comment Line—**
>
> **Source code line that is "pure" comment.**

Remember that a pure comment line starts with a semicolon or an asterisk. These are used to inform people. An active line usually *must* have either a valid op code for the 6502 or a pseudo-op for the assembler in it. "New" EDASM may interpret anything else as the name of a macro and will generate an error message if that macro does not exist.

You can immediately load and view an existing EDASM source code with *Apple Writer IIe*. And the source code will be readable. The only things missing are the line numbers and tabbing. We'll find out how to handle these shortly.

Since a word processor can put anything anywhere, the results of your entry or editing with *Apple Writer* may not be compatible with EDASM when you save your work. Since this is ungood, we have these use rules . . .

FOR EDASM COMPATIBILITY

A carriage return MUST be provided at the end of each source code line.

The text file saved to disk MUST NOT have any line numbers in it.

The file saved to disk MUST NOT have any "padding" spaces in it.

You might have used a word processor where you put carriage returns only at the end of each *paragraph* instead of at the end of each line. This is clearly a no-no if you are going to ask a free-form word processor to create files that must be accepted by a line-oriented editor or assembler.

So, be sure to have a carriage return at the end of each EDASM source code line. Lines usually must be eighty or fewer characters long.

We will shortly see how to add and remove line numbers from your word processor files. Note that on the final save to disk of your source code, you must have no line numbers and no padding spaces. (Note that "new" EDASM will allow imbedded tab commands.) On comment lines, your first character must be a semicolon or an asterisk, and anything goes after that. On active lines, you must start with a label, a IIe tab, or a space. The first space signifies a tab that moves you to the op code column. The second space signifies a tab that moves you over to the operand column. A final space, preferably followed by a semicolon, signifies that a tab is to take you to the comment column.

With "new" EDASM, you can imbed [I] tab commands into your source code, but note that you must not directly use the tab key to pad spaces into your text.

Note that the file on disk sees only single spaces, not tabs or groups of spaces used to fake a tab. The tabbing only gets done *after* EDASM reads the source code file from the diskette.

If you are using a different assembler or a different word processor, be sure you know exactly what the code on disk looks like. Also, be

certain that both your assembler and word processor can talk to each other.

LINE NUMBERS

Most editors of most word processors are line oriented. This means there has to be some way to point to a certain line. Line numbers are the obvious way this is done. As we've seen, you normally number your source code lines from 1 to N in order, with nothing skipped and nothing out of order.

But, with a "free-form" or screen-oriented word processor, you can see the order the lines are in, and it is usually obvious which line is which. So, line numbers are not particularly needed for most entry and editing of most source code done the new way.

In fact, with one or two exceptions, line numbers are totally unneeded and are a royal pain for "new way" editing. One exception takes place on an editing where you are correcting errors such as ILLEGAL OP CODE LINE 187. In this case, we sure would like to be certain we are really "fixing" the problem the assembler thought was on line 187, rather than fixing something that ain't broke.

What would really be nice is to be able to turn the line numbers on and off at will. Nicer still, let's keep any old line numbers the way they are until we are done with them, rather than renumbering continuously and automatically. This way, any additions or corrections to your source code won't mess up the numbering that was on the *previous* assembly listing.

So . . .

> **Line numbers are not needed most of the time with "new way" editing.**
>
> **It is best to be able to turn any line numbers on and off at will.**
>
> **A supervisory language with NUMBER, RENUMBER, and UNNUMBER commands is an ideal way to handle this.**

You won't need line numbers much in your new way source code editing. When you do use them, you can keep them the way they were for as long as you like, even if there are temporarily 15 new and nonnumbered lines between old line 258 and old line 259.

What you need is three magic pushbuttons. Press NUMBER to number all your source code lines. Press RENUMBER to cover any new entries. And press UNNUMBER before you save your source code to disk, or just to unclutter the screen.

Well, there's white magic, and there's black magic, and then there's WPL.

If you haven't met this dude, WPL is short for *word processing language* and is utterly astonishing. What WPL does is automate word processor use, much the same way an EXEC file automates program execution. WPL is built into both *Apple Writer II* and *Apple Writer IIe*. Just about anything you can command from the keyboard, WPL can do automatically and by itself.

The intended uses of WPL are for such things as automatic form letters, doing word counts, mailing list management, multicopy printing, menu or prompt generators, and lots of other text-handling nasties that took special effort before.

It is very easy to get a love/hate relationship going with WPL. This language (it's really just an interpreter) is very lopsided. For instance, WPL is probably one of the most powerful editing languages available anywhere ever. A command to take a very long string full of carriage returns and commas and manipulate it six ways from Sunday is fast and trivial. But a simple multiplication by say 7/3 will take you some 58 *seconds.*

And those are 100 percent genuine American seconds, too, besides. None of your pansy milliseconds or microseconds here. No siree.

What really blows everything else out of the water is when you use WPL to process *pictures* rather than words. The odd couple of WPL and HPGL does some truly astounding things. But, the *whipple glipple* machine is another story for another time.

Is it ever.

Anyway, it is very easy to build three programs called WPL.NUMBER, WPL.RENUMBER, and WPL.UNNUMBER that will fairly quickly put numbers into your source code under *Apple Writer II* or *IIe,* update them, or change them. The speed is not outstanding, but it is liveable. For instance, it takes around six seconds to number 100 lines of source code, and around four seconds for removal. The longer the code, naturally, the longer the time.

The way you use a WPL program is to have a copy of the program on disk. Then, from *Apple Writer,* you simply type ''[P] DO WPL.NUMBER,'' and away it goes. You start from *Apple Writer* and end up in *Apple Writer* a few seconds later. In interests of sanity, you probably will want to rename WPL.NUMBER simply as N, so that a ''[P] DO N'' does the job.

That simple and that quick.

The rules for WPL seem fairly complicated at first. But all you have to know is that a WPL program is a string of letters, symbols, and characters. It's just like anything else you would write on a word processor. If you don't want to key in the several dozen keystrokes involved, you can get these programs ready to go on the companion diskette.

Let's look at the three programs . . .

Number

To keep things simple, we will start our line numbers at 1001. This keeps the number of digits in use at a constant four. Obviously, you can mentally subtract 1000 from the *Apple Writer* number to get the EDASM number.

Brace yourself. WPL code looks awful at first glance. Anyway, here is the numbering program . . .

```
***********************************************************************
*                                                                     *
*                        WPL.NUMBER                                    *
*.....................................................................*
*                                                                     *
*                                                                     *
*       pgost                                                         *
*       z psx+1                                                       *
*       f<><>(x)- . <                                                 *
*       y?                                                            *
*       pgoz                                                          *
*       dn d                                                          *
*       f<>=. <><                                                     *
*       y?                                                            *
*       p                                                             *
*       b                                                             *
*       f<><<                                                         *
*       y?                                                            *
*       p                                                             *
*       pqt                                                           *
*       st pnd                                                        *
*       ppr [L]                                                       *
*       p            { number a source code file }                   *
*       psx1000                                                       *
*       b                                                             *
*       f<<><                                                         *
*       y?                                                            *
*       p                                                             *
*       b                                                             *
*       ppr                                                           *
*       ppr                                                           *
*       ppr            ***** busy - please wait *****                 *
*       pgoz                                                          *
*                                                                     *
***********************************************************************
```

Uh. Huh?

Let's try it in English instead. It goes something like this . . .

Please go to the label "st" so the part of the code that has to run fast can come first on the listing.

Next, please clear the screen and turn off the display so things run much faster. Then set the line number to 1000.

Go to the beginning of the source code and temporarily add a new carriage return at the start.

Put a busy message on the screen since this will take a while. Now go to label "z" to actually enter the numbers.

We are ready to start from the beginning of the source code. Add one to the present line number.

Next, find the next carriage return and replace it with a new carriage return, the number, a dash, a space, and a period. Repeat this till you run out of source code. Then go to the label "dn."

To finish up, remove the last line number since it is beyond the last source code line. Then, go to the beginning, and remove the temporary first carriage return.

Then, quit.

As you can see, WPL commands sure are shorter than English text is!

The game plan here is to turn off the display and replace it with a

busy message. Then you add a new carriage return at the beginning and set your first line number to 1001. Each carriage return gets a line number placed after it. The final line number is deleted, as is the extra carriage return at the beginning . . .

You can rearrange things to suit yourself if you like. I've set this up so the number is followed by a dash and then a space, then a period and a final space. This looks fairly good, but you might like to handle it differently.

To give you a hint of what is going on in WPL, the first string of letters is a label, if used. This label identifies the line. A single space is used to get from the label field to the "op code" field. The character in the op code field is interpreted as a control character. Thus, a command of "b" really means "[B]," which means "Go to the start of the file." Characters following the "op code" control character are ordinary ASCII characters as you would enter from the keyboard. The "<" is a searching delimiter, while ">" substitutes for "carriage return" during a search or replacement.

Note that the [L] in "ppr [L]" means to use "<CTRL> L" instead.

More details, of course, in the WPL manual. If you have not yet gotten involved with WPL, you are missing a biggy. Get with it.

At any rate, all you have to do is activate WPL.NUMBER, and line numbers magically appear in your source code a few seconds later.

Here's how you go the other way . . .

```
*****************************************************************
*                                                               *
*                      WPL.UNUMBER                               *
*...............................................................*
*                                                               *
*                                                               *
*        pnd   { unnumber a source code file }                  *
*        b                                                       *
*        ppr [L]                                                 *
*        ppr                  ***** busy - please wait *****     *
*        f<<><                                                   *
*        y?                                                      *
*        b                                                       *
*        f<>????- . <><a                                         *
*        p                                                       *
*        b                                                       *
*        f<><<                                                   *
*        y?                                                      *
*        pqt                                                     *
*                                                               *
*****************************************************************
```

As you can see, unnumbering is faster and easier than numbering. Here's the same thing in English . . .

Please turn off the display. Go to the beginning and clear the screen. Then put a busy prompt down since this will take a while.

Put a temporary carriage return here. Then return to the beginning.

Next, go through the source code file and find a carriage return followed by any four numerals, followed by a dash, a space, a period, and another space, a period, and another space. Replace all this with a single carriage return.

Continue this replacement til you run out of source code. Then go back to the beginning and remove the extra carriage return.

Then quit.

All this program does is find every carriage return followed by a four-digit line number and replace it with only the carriage return.

Oh yes, one very important gotcha. There are no labels in this particular WPL program, so you *must* type a space before each and every entry.

Renumber

We won't show you the code for WPL.RENUMBER here. It is on the companion diskette, all ready for your use. All WPL.RENUMBER does is first use the WPL.UNNUMBER code to erase all the old line numbers, and then uses the WPL.NUMBER code to give you the new ones. Just bolt the two programs together to do your own WPL.RENUMBER.

Normally, you will leave all the line numbers off your source code during entry and editing. The only time you would want them is when you are making specific corrections per the error messages on an assembly listing.

Here are the suggested use rules . . .

SOURCE CODE NUMBERING
Do NOT use line numbers during normal source code entry and editing.
Add line numbers only when fixing code pointed to by error messages.
Renumber only after you have fixed all the old errors.
REMOVE ALL LINE NUMBERS BEFORE SAVING YOUR SOURCE CODE TO DISK!

Some practice will make it all obvious. The longer you work with source code, the less often you will need or use line numbers. You might also find line numbers handy, but not essential, for COPY and MOVE actions.

TO TAB OR NOT TO TAB?

That is the geniffletravitz.

Or is it? Time out for a side trip before we get into tabbing.

One of the other projects around here is a super-cheap data acquisition system that uses EEPROMS. The idea is to use an EEPROM just like you use a cassette in a tape recorder. You want to have something tiny and ultra cheap that can be buried for a month to pick up stream levels, weather conditions, or just about anything ecological, biologi-

cal, speleological, or heaven knows what elseological, and do so at 1/1000th the usual price. After logging, you haul the EEPROMs off to your nearest Apple for disk storage, printing, plotting, telephone transmission, or whatever.

Fine. The thing is so simple it doesn't even have a microprocessor or a keyboard or a display. Only a counter, an A/D, the EEPROM, a clock, a "wake-up" circuit and a transistor radio battery. At a cost of $20, instead of $20,000.

Comes time to use it, all the users want to know how to identify the data on the EEPROM. Lots of expensive solutions were suggested, involving keyboards, fancy switches, displays, data blocks inside the EEPROM, and all sorts of other unworkable schemes.

Then one user came up with "How about using adhesive tape and a ball point pen?" Which not only is totally obvious and ridiculously cheap, but it is also far better and far more flexible than any of the other methods.

The point, of course, is . . .

DON'T SWEAT THE SMALL STUFF!

Tabbing is small stuff.

Now, ideally, you would like to enter your source code in exactly the same way that it is going to appear on the final assembly listing. While editing or changing, you would like the formatting to stay the same. This means that comments stay untabbed as comments, and that active lines get properly tabbed into their various fields.

One really nasty "feature" of EDASM is that it tabs the comment lines during editing. Which makes your comments totally illegible and causes all sorts of hassles. One very hard thing to do is add a new line inside a comment box. So, done the "old way" with EDASM, comment lines were a royal pain, because you never were sure how they were going to turn out.

So, if you tab comments, you get royal problems. On the other hand, if you do not tab the active fields, all you get is a slightly ratty display during entry and editing. Label, op code, and operand fields get separated by a single space. This looks awful, but you easily get used to it, and it doesn't particularly contribute to errors or lead to other problems. Naturally, your final assembler listings will be properly tabbed.

Which brings us to this . . .

If you have to pick between tabbing everything and tabbing nothing, then tab nothing.

Now, tabbing on a IIe is trivial. Just set up your tab file and away you go. The problem comes up when you read existing source code that lacks tabs or space padding, or else when you have to "untab" to save your source code back to disk. Ideally, you should tab and untab only the active lines and not change the comment lines.

All of this weaseling around because I haven't yet found a really

good way for you to selectively tab and untab while "new way" writing source code.

Yes, you can write a WPL program to selectively tab, but this route is painfully slow. And the timing is nonlinear, which means that long source codes take even longer than you would expect.

Forever even.

The next most logical thing, of course, is to modify WPL. I've done this for me, but it's not quite ready for you. Adding a new WPL command of "put the cursed character into the $d string" brings timing down into the "just barely unacceptable" range. The best way is to link your own machine language selective tabber to WPL.

Which, of course, we will save as an excercise for the student.

"New" EDASM does ease the tab and untab problems somewhat, since you can imbed unique [I] tab symbols rather than spaces. We will put the best solution on the companion diskette for you to play with.

Seriously, selective tabbing is small stuff. Don't sweat it. Think ballpoint and adhesive tape. Untabbed entries into "new way" source code look a little odd, but you get used to them, and they don't encourage errors or sloppiness.

TRYING IT

The simplest way to play with "new way" source code writing is to take some "old way" source code, load it into your word processor, and play with it. Once you start doing things the new way, you will never go back.

Except possibly for trivial fixes and quick changes.

One suggestion. Take the mystery program from the last chapter and enter it as a word processor text file. Use the spaces as spaces, and don't tab. Then save to disk and assemble per the details of the next chapter. Check this against your original "old way" code.

Note that some of the procedures involved in "new way" editing will be slightly different. For instance, instead of NEW and ADD, you'd use [N] Y. There'd be no [Q] for quit, and your SAVE would get replaced with [S] MYFIRST.SOURCE. All you are doing here is substituting word processor commands for editor commands, one on one.

Both the source code and the assembly results should be identical.

DIFFERENCE BETWEEN MY "NEW WAY" EDITING AND YOURS!

6

ASSEMBLING SOURCE CODE
INTO OBJECT CODE

Writing your own source code is only half of the game. What you
are really after is usable and runnable object code. The assembler half
of an editor/assembler package gets you from source to object . . .

> **The assembly process takes an already
> existing source code and uses it to
> generate working object code.**

On a disk-based assembler, such as EDASM, the source code is usu-
ally read from disk and the object code is usually generated to disk.

Remember that source code is a series of scriptlike instructions
more or less in English, while object code is your final ready-to-run
machine language program. Remember also that source code is often
a text file, while object code is nearly always a binary file. Both, of
course, must have different names.

Assembly takes place in two or more passes. The source code is first
read and all of the labels and label references are put into a suitable
symbol table. Then, once all labels are known and understood, a sec-
ond pass does the actual assembly, creating the object code to disk for
you.

The entire source code is usually *not* read into RAM all at once.
Instead, the source code is read one line at a time as needed. All that
has to go into memory is a list of labels and other references. Thus,
you can assemble a source code that is much longer than the room

you have in RAM. With the CHAIN command, you can link as many source codes together as you want. You can also use "new way" source codes that are so long you cannot "old way" edit them, and yet still be able to do an assembly.

There are several key rules you must obey before you do an EDASM assembly . . .

EDASM ASSEMBLY RULES

The source code MUST be previously saved to disk!

The PR#1 command MUST precede an ASM command for a hard copy assembly listing. ALWAYS use this listing.

The diskette in the selected SLot and DRive MUST have the ASSM module and an unlocked ASMIDSTAMP module on it.

Once again, EDASM is usually a disk-based assembler. Anything in RAM gets overwritten when you begin any assembly process. Thus, you *must* have previously saved your source code to disk, or you will lose it forever. This is needed even when doing "in-place" assembly with "new" EDASM.

As we've already seen, it is *always* a good idea to generate a hard copy assembly listing. If you fail to do this, your object code may end up different from your printed documentation, which leads to lots of trouble.

You only get a hard copy of your assembler listing if you have previously given the PR#1 command. Fail to do this and you assemble without any hard copy listing. The PR#1 command will stay in EDASM so long as you do not exit the program or remove power. Stopping to test your object code will end EDASM, and a new PR#1 will be needed on the next go-round.

The EDASM assembly process is set up to go to the previously set SLot and DRive to find the assembler code and the ID stamp. Default values here are the usual slot 6, drive 1. The ASMIDSTAMP must be kept unlocked at all times.

The source code does *not* have to be in this drive when you begin assembly, as you will have a later chance to swap disks with a "HIT ANY KEY TO CONTINUE" prompt.

Let's look at some . . .

ASSEMBLER COMMANDS

Assembler commands are the keystrokes needed to start or continue the assembly process. The most important of these is the ASM command, which is used to carry out the assembly process. Before using this command, though, be sure your source code is saved to disk, and

that you have given the PR#1 command to turn on the assembler listing, and that you have a disk in the selected drive that has both the ASSM module and an unlocked ASMIDSTAMP on it.

Let's check into these commands . . .

ASM

The ASM command is an abbreviation for *assemble,* and is used when we want to begin the process of converting the source code script into an object code machine language binary file.

ASM must be followed by parameters. First, you tell ASM the name of the source code. Then you add a comma, and tell ASM the name of the object code file to be generated. Then, if needed, you tell EDASM which slot and which drive to use.

For instance, we might use . . .

ASM ZILCH.SOURCE,ZILCH,S6,D2

. . . as our assembly command. This says to "Assemble the source code named ZILCH.SOURCE in (or soon to be in) the default slot and drive. Create an object code named ZILCH using slot 6 and drive 2."

There are several options here. You do not have to specify a slot and drive if they do not change. You have two ways to name your source and object files . . .

TWO WAYS TO NAME FILES
A command of ASM ZILCH will create an object file named ZILCH.OBJ0.
The ASM ZILCH.SOURCE,ZILCH command will create an object file named ZILCH.

Thus, if you fail to give EDASM the name for an object file, it will automatically tack a ".OBJ0" onto your source code name for the object code. Should you have several absolute ORG pseudo-ops in your source code, each separate object code will be successively numbered with .OBJ0, .OBJ1, .OBJ2 . . . trailers.

On the other hand, if you properly tell EDASM, it will name your object file anything you like. But only one absolute ORG should be used per source file if you do things this way.

I prefer to end all my source codes with a ".SOURCE" trailer. This identifies all source codes as source code, and your object code ends up with a reasonable "ready to run" name.

Once again, your source and object codes must all have different yet related names. Version numbers should be used, and you should *never* overwrite an old source code.

EDASM is very fussy over what goes where in the ASM command, so you have to pad with extra commas if you skip something in this command. For instance, "ASM ZILCH,D2" will create an object code

named "D2." Just what you always wanted. Instead, use the "ASM ZILCH,,,D2" command to change drives.

When the ASM process begins, it goes to the previously set SLot and DRive and gets *only* the ASSM module and the ID stamp. It then rewrites the next version number back to the ID stamp. After that, it stops and gives you a "HIT ANY KEY TO CONTINUE" prompt.

This prompt gives you a chance to swap diskettes. You would do this to be certain your source code is now sitting in the selected drive. This is handy if you only have a single drive, or if your source code diskettes do not have the EDASM modules on them.

I like to use two drives and always keep copies of all EDASM modules on all the source code disks. This is better and simpler, and the key prompt defaults to a nuisance.

The "standard" way of using EDASM is to have two disk drives. A diskette with a copy of EDASM goes into drive 1. Your source and object code goes into drive 2. Before assembly, you do a PR#1 and a DR2 command. The assembly command will then go to drive number one to get the assembler. It then asks for the key prompt. After that, the source code is read from drive number two, and the object code is sent to drive number two.

The "substandard" way of using EDASM is to have one disk drive and to keep a copy of all the EDASM modules on the same diskette as holds your source and object code.

If you don't like the "standard" or "substandard" way, then just swap the diskettes around on the HIT ANY KEY TO CONTINUE prompt, and use DR1 and DR2, along with trailing commas, as needed.

By the way, if you are always using the "standard" or "substandard" setup and if you know how to modify a machine language program, here is how to get rid of the HIT ANY KEY TO CONTINUE prompt on "old" EDASM . . .

```
TO PERMANENTLY ELIMINATE "OLD"
EDASM'S "HIT ANY KEY" PROMPT—

] BLOAD ASSM
] POKE 9202,96
] UNLOCK ASSM
] BSAVE ASSM, A$1200, L$22FB
] LOCK ASSM
```

Naturally, don't do this on your DOS TOOLKIT diskette. Do it on a backup copy or one of your working assembler diskettes. Someday you might need the prompt for something fancy, or if one of your drives goes sour on you. This patch is for "old" EDASM *only!*

[C] *(STOP!)*

Sometimes you would like to stop an assembly in the middle, because of some very obvious errors, a sick printer, or whatever. The "[C]" command aborts the assembly and restores everything back the way it was. It does so in an orderly and safe process.

An important rule . . .

> **Do NOT use "RESET" to stop an assembly in process!**
>
> **Always use "[C]" instead.**

The reason to not reset your way out of an assembly is that disk text files may be open and you might be in the process of a disk read or write that could plow the diskette as well as the program. Always use this "[C]" for an orderly exit. The controlled exit takes care of shutting things down in a safe and reasonable order.

If you do abort an assembly, there may be an illegal and nonworking object code file remaining on your diskette. Be sure to delete this or it will return to haunt you later.

SPACE

The spacebar may be used to temporarily halt an assembly. Each successive spacebar hit will then assemble one line of object code at a time. This is intended to let you assemble to screen and see what is going on. On "new" EDASM, assembly resumes at full speed when any other key is pressed.

But, as we have seen, you should always have your printer on and should always be making an assembly listing when you are assembling. So, this spacebar use is not recommended.

Almost certainly, your first assembly attempt will do nothing but generate a bunch of error messages. Guaranteed. Before we look at these messages and see what they mean, let's look at . . .

ASSEMBLY LISTINGS

If you remembered to turn your printer on and didn't forget to give a PR#1 command before trying an assembly, you will generate an *assembly listing.* The assembly listing is your hard copy documentation. It gives you *both* source code and object code in one place at one time.

Here's how to read an assembly listing . . .

HOW TO READ AN ASSEMBLY LISTING:

ADDRESS	HEX BYTES	LINE	LABEL	OPCODE	OPERAND	COMMENTS
8CD0:	A2 27	254	ROWDUN	LDX	# $27	;HANDLE SPECIAL MAPPINGS
8CD2:	BD 80 40	255	EVNTHD	LDA	$ 4080,X	;8 TO 7
8CD5:	9D 00 3C	256		STA	$ 3C00,X	;

THE **ADDRESS** IS THE HEX LOCATION WHERE THE **FIRST** OBJECT CODE BYTE FOR THIS LINE IS TO GO. ONLY ONE 8-BIT BYTE ALLOWED PER MEMORY LOCATION.

THE **LABEL**, IF USED NAMES THE LINE SO THE ASSEMBLER CAN FIND IT FOR JUMPS, BRANCHES, OR OTHER REFERENCES.

THE **COMMENTS**, IF USED, HOLD REMARKS USEFUL TO THE USER OR PROGRAMMER, BUT OTHERWISE IGNORED BY THE ASSEMBLER. SOME LINES MAY BE ENTIRELY COMMENTS.

THE **HEX BYTES** ARE THE ACTUAL OBJECT CODE ASSEMBLED DURING THIS LINE. ONE, TWO, OR THREE BYTES WILL BE GENERATED, DEPENDING ON THE ADDRESS MODE.

THE **OPCODE** IS THE 6502 INSTRUCTION THAT THE ASSEMBLER IS CURRENTLY CONVERTING INTO HEX BYTES OF OBJECT CODE.

THE **OBJECT CODE** GENERATED DURING THE ASSEMBLY ARE THE VALUES IN THE HEX BYTES COLUMN. ONLY THE HEX BYTES GO INTO THE OBJECT CODE!

THE **LINE NUMBER** TELLS US WHICH LINE OF SOURCE CODE IS PRESENTLY BEING ASSEMBLED.

THE **OPERAND** QUALIFIES THE OPCODE AND PICKS THE ADDRESSING MODE IN USE. THIS OPERAND CAN BE A VALUE, AN ADDRESS, OR A LABEL.

HERE'S THE **HEX DUMP** OF THE OBJECT CODE CREATED BY THIS EXAMPLE:

8CD0: A2 27 BD 80 40 9D 00 3C

As you can see, an assembly listing looks more or less like source code, only there are more fields. An EDASM assembly listing has seven or more fields. These are the *address* field, the *hex bytes* field, the *line* field, the *label* field, the *op code* field, the *operand* field, and finally the *comment* field.

The last five of these are the same as the fields in your source code.

The address field holds the *beginning* address of the line being assembled. Remember that the assembly process consists of starting at some ORG origin in memory and then using a "pocket card" to convert all the mnemonic op codes into working machine language.

For instance, say we have an ORG $0800, and want to do a STA $877D as our first "real" op code. We immediately see that the addressing mode is absolute, since we have four hex digits and no commas or parentheses. Checking our pocket card, we see that we need an $8D byte for STA absolute, so we put an $8D into memory location $0800.

More correctly, we put an $8D into a binary file on diskette, whose eventual running position is to be $0800 in memory.

An absolute store needs three bytes to be completed, since we need the specific op code, the position on the memory page we are to store to, and finally, the page we are to store to. The 6502 always puts the position before the page, so the next two bytes go in reverse order of $7D $87.

A hex dump of what we have done so far would look like this . . .

$0800– 8D 7D 87

That's assuming you took the object code and loaded it into the Apple. But, remember, we are only creating a disk file at the present

time. Nothing is being put in specific "ready-to-run" Apple locations just yet. It all normally goes onto the disk.

That single line of source code generated three bytes of object code for us, since STA absolute is a three-byte instruction. The next instruction has to start at location $0803, since locations $0800, $0801, and $0802 already got used up. Remember that each memory location can hold one, and only one, 8-bit byte.

So, the address field will show you the beginning address of what is being assembled at this point in time by this line of source code. If you are on a comment line, the address will not change. It stays the same until some new bytes of object code actually get generated.

The hex bytes field will show you the machine language bytes being generated by the present line of source code. You will get one, two, or three bytes, depending on the instruction and the address mode you have selected.

Should you be generating some nonrunning bytes such as a stash or a file, EDASM still puts a maximum of three bytes in the hex bytes field. For instance, an 8-byte file defined using a DFB source code will show up as three successive lines on your assembly listing. You'll find three bytes generated in the first two lines and two bytes in the last.

The remaining five fields of your assembly listing are the same as those of your source code. The line number tells us which line of source code we are working on. The label field points to that line and is used as a reference any time the assembler has to find this line or the meaning of this label. The op code field holds a 6502 op code or else an EDASM pseudo-op intended to go only as far as the assembler. The operand field qualifies the op code as needed, selecting an address mode, the number system in use, and choosing between addresses, fixed values, and labels.

The comment field, of course, is reserved for comments.

You can also have comments that use up an entire line of source code. These are intended for people. The assembly process does not generate any new bytes of object code on a comment line. The address stays the same and gets carried down to the next source code line.

Normally, you won't have those neatly labeled column headers we have just shown you. The function of each column of assembly listing becomes obvious when you study it for a while. Other assemblers may have these columns in a different order.

On "new" EDASM, you will also find a column on the assembly listing for "branch taken" addresses. You can also optionally turn on or off a column of execution times. This latter feature is most useful for time critical code.

Let's look at a specific example. Remember the "mystery" program from back in chapter four? You should have it saved on disk under the name MYFIRST.SOURCE by now. In fact, you should have done this both the "old way" and the "new way." You may have two source code files if you remembered to give the second one a different name.

We'll use the "substandard" assembly method.

Boot EDASM from drive 1 and fix the ID stamp any way you want to. Then, when the colon comes up, type . . .

```
PR#1 <cr>

ASM MYFIRST.SOURCE,MYFIRST <cr>
```

The drive should start whirring away, picking up the ASSM module and incrementing the ID stamp. Then, you should get the HIT ANY KEY TO CONTINUE prompt. If it isn't already there, put the diskette with MYFIRST.SOURCE into drive 1 of slot 6. Be sure there is enough room on this diskette for the MYFIRST object code. If a MYFIRST file is already there, make sure it is a binary file and that it is unlocked. Also make sure your printer is on and ready to go. Then, hit the spacebar.

The assembly process should begin, and three things should start to happen. The printer should give you an assembly listing; the video screen should "echo" what is going onto the printer; and some object code called MYFIRST should be getting generated onto disk.

When the assembly is complete, you will get another "HIT ANY KEY TO CONTINUE" prompt. This tells you to put a diskette that holds the ASSM modules back into drive 1 of slot 6. The EDASM package then "cleans up" its text file act and gently returns you "live" back into the editor half of EDASM. The disk drive will get very unhappy if you have the wrong diskette in it at this time.

When all is finished, type END to get out of EDASM, and then do a BRUN MYFIRST. If you haven't figured out yet what MYFIRST is supposed to do, it will become immediately obvious, assuming everything went OK. A total of eleven bytes of code should have been created.

Your mystery program sits between $0300 and $030A. You may want to inspect it after BLOADing or BRUNning. Remember that CALL -151 gets you into the monitor from Applesloth.

Here is what the assembly listing for MYFIRST should look like . . .

```
MYFIRST.SOURCE                          20 JUN 83 DEL #01 PAGE 1


----- NEXT OBJECT FILE NAME IS MYFIRST
0300:              3            ORG  $0300      ; BRUN CODE AT $0300

0300:              5 ; *************************
0300:              6 ; *                       *
0300:              7 ; * THIS IS MY VERY FIRST *
0300:              8 ; *     TRY AT EDASM      *
0300:              9 ; *                       *
0300:             10 ; *************************


0300:             12 ;        *** HOOKS ***

C030:             14 SPEAK   EQU  $C030      ; WHAP SPEAKER
FCA8:             15 WAIT    EQU  $FCA8      ; MONITOR TIME DELAY


0300:             17 ;        *** MAIN PROGRAM ***

0300:A9 22        19 BLATT   LDA  #$22       ; SET TONE VALUE
0302:20 A8 FC     20         JSR  WAIT       ; AND DELAY
0305:2C 30 C0     21         BIT  SPEAK      ; WHAP SPEAKER
0308:4C 00 03     22         JMP  BLATT      ; AND REPEAT



*** SUCCESSFUL ASSEMBLY: NO ERRORS
```

You should also get two pages of label lists, with BLATT, SPKR, and WAIT first in alphabetical order, and then in numeric order by value. To turn this listing off, you should have used a LST OFF pseudo-op as your last program line.

Let's see what you know about assembly listings—

1. Why are lines 1, 2, 4, 11, 13, 16, 18, 23 and 57 missing?
2. Why doesn't the address change in lines 5 through 10?
3. Why does WAIT appear in the label field of line 15, but appears instead in the operand field of line 20?
4. Why does the address field change weirdly in lines 12 through 17?
5. Why does the address change by two counts between lines 19 and 20, rather than the three counts it changes between lines 20 and 21?
6. What address mode is used on line 19? On line 20? On line 22? On line 23? On line 57?
7. Why are there no labels in the label fields of lines 20, 21, and 22? Are there any labels in use on these lines?
8. Why are there semicolons at the beginning of some lines and at the middle of other lines?
9. Why are some lines right against each other, while others are separated by white space?
10. How does line 22 know where to jump to?
11. Why did you get both page numbers and a label listing?
12. Why is the highest byte of the object code at $030A, yet the address field never gets past $0308?

If you can answer all of these, you are off to a very good start in understanding and using assembly listings. Both your own and those of others.

Before you run off all ecstatic over how easy assembly language is, though, we have to get into the rather ugly topic of . . .

ERROR MESSAGES

Your first attempt at any assembly is practically guaranteed to produce errors. Some of these will be very minor. Others will be major. Yet others will stop the assembly process dead in its tracks. A good assembler will try to tell you what you did wrong.

There are three kinds of assembly errors . . .

Fatal Error—

Something so bad that the assembly process stops completely.

Nonfatal Error—

Something bad but not bad enough to stop the assembly process.

Thinking Error—

Anything that stops perfectly assembled code from doing what is expected of it.

A fatal error is one so bad that the assembly process stops dead in its tracks. Most of these involve disk problems.

A nonfatal error lets the assembly process continue, but points out to you things the assembler did not understand, or that it took a guess on. A few bytes of worthless source code may have been generated in the process.

Now, why on earth would you want to continue assembling worthless code?

Because it is very easy to get dozens of nonfatal error messages. Many of these will be more or less trivial and easy to fix. Thus, you can often correct great heaping handfulls of nonfatal errors at one sitting. Otherwise, it would take a separate edit-assemble-debug round trip for each individual error. This would have to get done regardless of how trivial or minor the error is. Most painful.

As we've pointed out before, the message "***SUCCESSFUL ASSEMBLY: NO ERRORS" is just about meaningless. It does suggest strongly that your Apple did not sustain a direct meteor hit in the last few minutes, but that is about it.

Which brings us to those thinking errors. Thinking errors are faults in your logic that prevent the properly assembled object code from doing what you expect of it. This can be anything from not understanding address modes to trying to do the impossible. Naturally, neither EDASM nor any other assembler can second guess what it was that you thought that you might have really wanted.

Most assemblers have ways to tell you about fatal and nonfatal errors. But, obviously, they cannot second guess what you had in mind in the first place. Because of this, the *** SUCCESSFUL ASSEMBLY: NO ERRORS acts like a threshold. When you get to here, the really dumb, simple, and stupid mistakes are mostly behind.

Only the subtle, frustrating, and challenging errors remain.

Lots of round trips will be needed through the edit-assemble-test process. But, since you are going to make lots of mistakes anyway, you might as well get good at it . . .

> **Aim for not less than three error messages per line of source code.**
>
> **If you are going to foul up the works, then do it with class.**

Remember though, that mistakes are *absolutely essential* to creating decent assembly or machine language programs. It is only through mistakes that you progress, and only through mistakes that the real problems and the real opportunities become obvious.

We won't spend too much time here on thinking errors. That's best done elsewhere. Let's instead take a closer look at EDASM's fatal and nonfatal errors and see what they tell you and how to interpret them.

Fatal Errors

Most of EDASM's fatal errors involve disk access. If EDASM cannot get something off of a disk or cannot put something back on, a fatal error results, and the program must come to a grinding halt.

These errors are not spelled out by EDASM. Instead, the usual DOS error codes are used. Some of these are spelled out by DOS, while others are numbered only, and preceded by an OOPS! DOS ERROR!

CODE=XX. It depends whether DOS finds the mistake or whether EDASM does.

Here are the fatal error messages and their DOS codes . . .

EDASM FATAL ERROR MESSAGES	
Write Protected	—(04)
End of Data	—(05)
File Not Found	—(06)
I/O Error	—(08)
Disk Full	—(09)
File Locked	—(0A)
Syntax Error	—(0B)
No Buffers Available	—(0C)
File Type Mismatch	—(0D)
Program Too Large	—(0E)

These DOS 3.3e error messages are discussed in depth in the EDASM manual, but let's review them here. Once again, you will get the message spelled out if DOS 3.3e finds the error by itself, while you will only get the numbered OOPS! codes if EDASM finds the error.

The "missing" error numbers in this list are other DOS errors that are extremely unlikely to occur when you are doing assembly work.

Let's run down the list. WRITE PROTECTED usually means you have the wrong disk in the wrong drive. Since you must be able to write to the ASMIDSTAMP, you cannot use a write-protected disk to hold EDASM. You also cannot save source code or generate object code onto a write-protected diskette as well.

That END OF DATA message is an inconvenient bug. It is the normal way that "old" EDASM ends up a normal load of its source code. But always be suspicious if you get the END OF DATA message real fast, since this may mean that you have just opened a new, empty, and wrong source code text file.

FILE NOT FOUND means you have the wrong disk in the drive, or else have misspelled your filename. FILE TYPE MISMATCH means you tried to load a binary object code instead of your source code text file, or else are trying to save some source code to a name already used for a binary object file.

That dreaded I/O ERROR means that the drive didn't work and is usually bad news, particularly if there is a punk rock score that goes along with it. Try opening and closing the drive door. Make sure you have an initialized diskette in the drive. If you have to, check the drive speed. If you have really plowed the diskette, try fixing it with *Bag of Tricks* or something similar.

DISK FULL is obvious, while PROGRAM TOO LARGE means you did something *really* stupid. Dumb even.

FILE LOCKED should also be obvious. But remember that you should always keep everything locked, except for ASMIDSTAMP. Better to get an error message than to plow a year's work. SYNTAX ERROR means you did something dumb on a direct DOS command. Misspelling UNLOCK is one way to do this. NO BUFFERS AVAILABLE

is very unlikely but is bad news. See the EDASM manual for an explanation.

If you get any of these messages, you will have to stop immediately and repair the damage. It is always a good idea to do a fresh power-down restart after any fatal error.

There are additional error messages in both versions of "new" EDASM. See their respective manuals for full details.

Nonfatal Errors

As the assembly process continues, nonfatal errors may be generated. These errors are not bad enough to halt the assembly process completely, but they are bad enough that wrong object code will be created.

These nonfatal errors are spelled out on your assembly listing as they happen. They are also summarized at the end of your listing. Some errors may be caught *twice* since there are two passes to the assembly process. If this happens, the same error may be double listed. As a reminder, the assembly process does not stop on a nonfatal error. This lets you correct bunches of these errors at once, rather than needing a separate assembly for each and every problem that crops up.

The nonfatal error messages are not covered in the "old" EDASM manual. Let's look at them, more or less in order of likelihood . . .

BAD OP CODE

There are only two "good" kinds of op codes in "old" EDASM. These are legal 6502 mnemonics, and legal EDASM pseudo-ops. Anything else gives you this error message. Misspelling, using Sweet 16 or 65C02 or 68000 op codes in "old" EDASM, or putting anything else at all in your op code field will cause this error message. Unless, of course, your assembler is set up to handle these.

"New" EDASM does support Sweet 16 and 65C02 mnemonics. If you use the MACLIB command, "illegal" op codes will be treated as macro file names.

One very easy way a beginner can get this message is to put a tabbing space before a label. This magically "converts" a label into an op code. Another way is to leave the source code line entirely blank. Don't forget that you *always* must have an op code or pseudo-op in EDASM's op code field, unless your line is pure comment and preceded by the usual semicolon or asterisk.

BAD EXPRESSION

This one usually involves the operand field. A bad expression is an operand that does not compute.

For instance, try immediate loading a 16-bit value. Since 16 bits won't fit into an 8-bit memory location, you have a bad expression. Now, of course, you can load a 16-bit value as a *pair* of 8-bit bytes, using the "<" and ">" operators. You can also DW or DDB 16-bit values. But you can't stuff 16 bits into an 8-bit slot.

Another way to get a bad expression is to forget the dollar sign on your hex values. If a decimal number has a letter in it, the poor assembler gets all confused.

Having the wrong address mode will also create a bad expression error message. Remember that not all modes are available for all commands. One that beginners often try is page zero, indirect indexed by X. Which is a great addressing mode.

Except it does not exist on a 6502.

Forgetting the operand entirely when it is needed is another way to generate this error message. A JMP or a JSR must be told its 16-bit destination address.

If you try some operand arithmetic that is not kosher, you might also get this error message. The best rule here is "try it and see." If it flys, fine.

NO SUCH LABEL

This nonfatal error also centers on the operand column. If you forget to define a label, you will generate this message.

Misspelling also causes problems. And, there's that all important dollar sign again. Remember that a label can be any group of letters and numbers that starts with a letter and has no spaces in it.

For instance, an LDA F347 command will try to find a *label* named F347. If, instead, you want *hex* address $F347, then you have to include the dollar sign in your LDA $F347. Remember also that labels *must* be used for page zero addressing, and these labels must be EQU'd ahead of time.

Forgetting the dollar sign can truncate numbers. For instance, a LDA #6E will put an 06 in the accumulator. To get the full value, use LDA $6E.

DUPLICATE SYMBOL

You are only allowed to define an EDASM label *once* in *one* label field in your entire source code. You can do this label definition ahead of time as an EQU, during the "real" op codes to point to a certain code line, or afterwards with a DFB as part of a stash or a file.

If you reuse a symbol, the assembler gets confused. Only assemblers that let you separate global and local labels will let you reuse the same label twice. Even then, there will be special use rules. Local labels are permitted under "new" EDASM's macro capabilities.

ILLEGAL LABEL

You get an illegal label if you try spelling one with leading numbers, rather than letters, or have unallowable punctuation in your label.

Some assemblers refuse to accept op codes or single letters as labels. The only "old" EDASM label that is forbidden is a single "A." Anything else goes, subject to the usual label rules. "New" EDASM also restricts "X" and "Y."

BAD EQUATE

A bad equate takes place when you try to define a label into something it cannot be. For instance, a single byte EQU cannot be set to a 16-bit value.

You should not use the "#" symbol during an equate. EQU means nearly the same thing. If your label is to be used as an immediate

value later as an operand, then the "#" symbol goes into the *operand* and not the label.

An EQU will only work properly if its operand can be converted into a fixed and known 8-bit or 16-bit value. You can use operand arithmetic with an EQU, provided it refers only to other EQUs.

Remember that EQUs are set up *before* the assembly process, while DFBs are generated *during* assembly. Thus, DFBs don't exist at the time the EQUs are set up.

OVERFLOW

You can get an overflow error on relative branch calculations. Remember that a relative branch can only go +127 or −128 blocks from where it is sitting in the address space. If the line label that branch is seeking is too far away, the branch cannot be completed, and you get this error message.

There are at least five solutions to the overflow problem. One is to shorten or rearrange the code enough that the branch is in range. The second is to branch to a second branch to pick up somewhat more range, without sacrificing relocatability. The third is to branch to a jump and then go anywhere you want.

The fourth solution is to try complementary branches, changing BEQ for BNE, and reworking the code so it has a different structure.

The fifth solution is to use Sweet 16's new "long branch" command under "new" EDASM.

By the way, don't forget that a branch across a page boundary adds one clock cycle to your timing. And don't forget that the older 6502's jump indirect command has a bug in it that *never* crosses page boundaries. Thus, the 6C op code will not work in the expected manner if it is in either of the top two slots of any page. This bug is fixed in the 65C02.

ALSO RANS—

There are three nonfatal error messages that are so rare that you may never see them. The BAD CALLING PARAMETERS error happens when you try to use an illegal slot or drive number, or otherwise get fumble fingered during an editing or assembly command.

The ILLEGAL DESECT/DEND message is a specialized one involving dummy sections set aside for later code insertion. This sees little use by beginning programmers.

Finally, SYMBOL TABLE FULL means that you overfilled the machine during assembly. Note that the entire source code is *not* stored in your Apple when assembly takes place. The source code is read off disk a line at a time, and all the labels and other symbols are picked off and placed into a symbol table.

EDASM can, in theory, assemble a source code that is much longer than it can enter and edit in one piece. You can CHAIN many source code files together. Two passes per source code chunk are involved. The first generates the symbol table, and the second combines the values in the symbol table with a line-by-line reread of the source code to generate both object code and an assembler listing.

If you somehow manage to completely fill the entire symbol tables in all of RAM, you will generate this error message. But this error is

very unlikely, even for a humongous assembly project involving several diskettes full of source code.

Handling Errors

What do you do when you get bunches of errors during an assembly?

Getting mad at the assembler doesn't seem to help much. Painful as it seems, you'll have to admit that . . .

> **Assembly errors are all YOUR own fault, caused by YOUR stupidity or YOUR carelessness.**

So, if you did it, you can also undo it. The first thing, of course, is to adjust your emotions to the point where you are not mad, or frustrated, or embarrassed, or whatever.

Take a break if you are upset. Go kick your neighbor's dot matrix printer. Write a nasty letter to someone who used "data are" or "datum is" in a speech or a paper. Ridicule someone who still uses octal. Snicker behind a dino machine's back. If you are in a student lab, just smile proudly and pretend all those beeps are part of your new music synthesizer, rather than assembly error flags.

Or, if you are me, go quest a tinaja. Marijilda alone has some world class candidates in its innermost sanctum. So does Frey.

Errors beget errors if you let them. So, recognize that an error is a correctable problem. Not only correctable, but something expected and anticipated as well. View errors as process.

And progress.

Should you get a fatal error, you should try again right away. Do it now. Aim toward getting an assembly and its assembly listing at least completed, even if it is wrong.

When you get down to nonfatal errors, go over the list and re-edit your source code. As usual, it is best to edit backward from end to beginning. During this process, you may also want to clean up and improve your comments, page breaks, and pretty printing.

As you gain assembly experience, you'll find yourself making far fewer errors far less often. Learn from your mistakes, and profit by them.

Naturally, there is no point at all in trying your source code to see if it works when there are still error messages being created during assembly . . .

> **There is no point in trying your object code if you are still getting assembly error messages.**

Resist the temptation to "hand repair" a couple of obvious object code bytes to get something working. The patch will surely return to haunt you, as will the impatient attitude and sloppy thinking that goes with it.

So, there really is some value to that *** SUCCESSFUL ASSEMBLY:

NO ERRORS message after all. It means that all the obvious, dumb, and incredibly stupid mistakes are now out of the way. All that remains is the challenging problem of getting your object code to do what you expect of it.

DEBUGGING

After you have gotten your source code assembled into object code and have done so without any error messages, you will want to test your final code to see if it works.

Needless to say, it will not.

The process of getting object code to do what you expect of it is called *debugging* . . .

> **DEBUGGING—**
>
> **Anything needed to get your object code to do what you want it to.**

Make no mistake of it. Debugging is art, not science. How good you get at it and how effective your debugging approach gets decides how decent an assembly programmer you will become.

Debugging is normally done in steps . . .

> **Debugging is done by repeated round trips through the edit/assembly/test process.**

We could spend volumes on debugging. But, then all you would know is how I go about debugging my programs at the present time. Naturally, next week or next month I will have better ways to handle my programs. But, we are worried about you. You become a decent debugger through lots of hands-on experience, learning from your mistakes as you go on.

Let's look at some "first principles" of debugging . . .

```
┌─────────────────────────────────────────┐
│          FOR EFFECTIVE DEBUGGING         │
├─────────────────────────────────────────┤
│                                          │
│  ALWAYS expect trouble.                  │
│                                          │
│  ALWAYS assume you are nowhere           │
│  near where you think you are.           │
│                                          │
│  NEVER get in a hurry or try to do too   │
│  much at once.                           │
│                                          │
│  ALWAYS be willing to make simple        │
│  models or test simplified code.         │
│                                          │
│  NEVER fight your object code. It is     │
│  trying to help you.                     │
│                                          │
│  ALWAYS be willing to put much more      │
│  time and effort into your work than     │
│  you thought you originally needed.      │
│                                          │
│  ALWAYS beta test.                       │
│                                          │
└─────────────────────────────────────────┘
```

The list could go on for several more volumes. Remember that your first assembly attempt **will** be wrong. There is absolutely no doubt about it. If you think it's right, then you simply do not understand the problem.

Now, you can improve your odds greatly and can generate better code by remembering two key points. The first is that the *sooner* you start punching code into a machine, the *longer* the task will take.

The second is that the initial 90 percent of a computer task takes up the initial 90 percent of the available time. The final 10 percent of a computer task takes up the final 90 percent of the available time.

So always expect trouble. The code *will* be wrong and *will* need reworking. If you allow for this absolute inevitability ahead of time, your mind set and your work attitude will get you where you want to be much faster and much easier.

You always *will* make dumb and stupid mistakes. Things so dumb and stupid that you will be utterly amazed that the code did *anything* at all, let alone what you wanted it to. So, always assume that you aren't nearly where you think you are at any time.

Getting in a hurry or cutting corners never works. All it means is that your customers will find the bugs for you. Their response and attitude when they do this might not be completely to your advantage, to say the least.

You should always try simple models and independent tests to show you whether some coding idea is effective and will work. Be ready to try a different tack, or drop back to something simpler that will show you where the real problems lie.

Once your object code gets past a certain point, it will genuinely *try* to work and will be screaming new ideas at you. If only you will listen. Let the code show you the way. Remember that computers are dumber than people, but smarter than programmers.

You will never finish any decent assembly project in the time you allowed for it. Almost always, you will have forgotten something really

fundamental, or else the code will show you a newer and much better way to solve a bigger, more needed, and more general problem.

It is very important to have others test and evaluate your code. This goes by the fancy name of *beta testing,* and is another must. Beta testing lets someone else's mind work over your creation. Since your thought processes, needs, and attitudes are different from others, beta testing is certain to improve your product. It's not creative unless it sells.

This one needs its own box for sure . . .

```
ANY DECENT COMPUTER PROGRAM
IS NEVER FULLY DEBUGGED.

NOR CAN IT EVER BE.
```

All an apparently working program tells you is that only the more blatant bugs have been removed from only the most obvious ways of using the program. But, rest assured, there are *always* bugs lying in deep cover.

Just waiting.

In fact, the number of bugs in a program can actually go *up* with time if you are not careful. This happens if you make repairs and patches in oddball ways to fix obvious things. These repairs make subtle changes in unexpected times and places. Someday . . . blooey.

As an example, countless versions of custom DOS 3.3 are available today. Many of these overwrite the INIT portion of the DOS code. But, parts of this overwritten INIT code are used in subtle ways by unexpected portions of DOS. Thus, practically all of the new "super DOS" programs have an armed bombshell sitting inside of them, waiting patiently.

Tick. Tick. Tick. Just don't press that red but . . . !

The usual way to start testing object code is first to load it into the machine. With some assemblers, you can have the assembler and your final code side by side in the machine at the same time. But this very much restricts how long your code can be, where it is located, and how you can interact with it.

Instead, more often than not, you will separately load and test your object code. This lets you test any code anywhere in the machine without any big restrictions. "Old" EDASM demands separate testing of this type, while "new" EDASM optionally lets you do in-place assembly.

If you are cross assembling or down loading, you will have to go through a song and dance to get the code compatible with the machine it is to run on. You might do this by generating a cassette tape to the other machine's standards, by using the Apple cassette hardware. Or, you might simply punch in the new object code to the target machine by hand loading it.

One heavy that will work to change diskette formats and leap other incompatibilities with a single bound involves using a serial port between the Apple and the target machine. Set up the Apple to send out serial ASCII. Set up the other machine to receive serial ASCII.

Transfer the code and put it in its new formats. It is an involved process, but it works and is simple to use.

Anyway, after you have gotten your code into the machine, list it to make sure it is what you think it is and is sitting where you think it does. A hard copy dump at this time is a very good idea. Be particularly on the lookout for oddball address modes used strange ways, and the usual question marks denoting illegal op codes.

Needless to say, if you can't get the code to list the way you want it to, it won't run either. Back to square one.

Do not pass GO. Do not collect $200.

I like to think of two stages in the debugging process. The dividing point between them might be called the *viability threshold* . . .

```
VIABILITY THRESHOLD—

That point at which defective code starts
doing something at least more or less
recognizable.
```

This is kinda like the difference between a pile of auto parts and a broken car.

The pile of parts is just so much junk. The broken car is both capable of being repaired, and even now, does certain carlike things. For instance, you can listen to the radio in a car with a broken radiator. But a radio lying on a pile of junk won't work by itself, since it has no battery or antenna.

Before you cross the viability threshold, you have to use sledgehammer techniques to find out what is wrong and how to fix it. After you cross the viability threshold, the code itself will help you along and will show you the correct way to go. Totally different, and often more subtle, debugging schemes will be needed above the threshold.

This "two-stage" debugging process is most obvious once you look for it. Just as soon as you can, let the code help you as much as possible. You will always know when you have crossed the threshold.

Let's look at some examples of "stage-one" debugging. Here the Apple just bombs or goes hooping off into nowhere, or else does things totally weird and totally unrelated to what you expect of the code.

The "stage-one" debugging techniques include . . .

```
                "STAGE-ONE" DEBUGGING

                Listing
                Hex dumping
                Single stepping
                Tracing
                Forced returns

                Traps
                Breakpoints
                Interaction
                Simplified models
                Falling back
```

Let's quickly review these.

Listing is doing an op code by op code dump of the program. Hex dumping gives you a printout of which bytes are sitting where in which order. Listing only works on legal code when done from a legal starting point. Dumping works any place and any time, on working code, files, or even abject garbage.

Single stepping is a monitor routine that fakes running your object code one line at a time. There is a fine single stepper in the old monitor ROM of the older Apples. RAM based copies of single-steppers are available from most user groups as public domain software, besides being commercially available in various programmer utilities.

But, better yet, there is a super new BUGBYTER debugging aid that comes free with either version of "new" EDASM.

Tracing is repeated single stepping at a fairly slow rate. This is best done to a printer, and you will get a printed record of what codes got done in which order, sort of an "audit" of what happened. But tracing can get old fast inside delay loops. Tracing also will not work on such things as screen clears, since the tracing process interacts with the clearing code and hangs the machine. BUGBYTER can get around this problem several ways.

One useful debugging technique involves forced subroutine returns. What you do is take your subroutines, and, one at a time, replace the first byte of each sub with an immediate return. This is otherwise known as hex $60. What forced returns do is separate whether the problem is in the subroutine or in the main code. Another use of forced returns is to bypass parts of the code that will take forever to trace or will hang the trace process.

A trap is a code line that jumps to itself, such as $3C56— 4C 56 3C. This translates to "when you get to me, go to me." Your Apple will get to this line and then "stick" there until you hit reset or turn the power off. You use traps to stop the machine at a certain point in your code.

Sometimes, you will not even get to the trap. Which says, that the problem is above the trap. Move the trap further up, dividing and conquering as you go along. When you hit the trap rather than the problem, you have isolated the bug.

Breakpoints are a formalized way to do traps. What you do is put an $00 or BRK into your code. When and if your Apple gets to this point in the program, an immediate interrupt is forced by the CPU. Depending on the age of your Apple and how you set the vectors, this BRK interrupt can return you to the monitor or else to diagnostic code of your choice. Once again, BUGBYTER handles this level of debugging beautifully.

With interaction, you use known good code to test new stuff. Test each piece of the target code separately and as simply as possible. This should isolate individual problems one at a time.

By the way, there is usually more than one problem remaining. And the second one can mask the first one, and vice versa. So, never assume that something "has" to be happening a certain way. Chances are that several bugs are involved at once.

By simplified models, we mean to set aside temporarily the program that is causing the problems, and make a simplified test of a simplified model of that part of the code that seems to be creating the worst of the problems.

Falling back involves throwing in the towel, and trying something

simpler, or else going back and rerunning and reusing the last good working code you had on hand.

If the program seems to be *really* weird and refuses to do anything rational at all, there are five important things to check . . .

WEIRDNESS CHECKS—

—Is the stack behaving?

—Did you get into decimal mode?

—Are registers and variables being used for two different things at once?

—Are the interrupts being handled correctly?

—Is an illegal op code hanging the machine?

The first weirdness check involves the stack. What you put on the stack must come off, and in the expected order. More pushes than pulls, more pulls than pushes, or using the stack contents in the wrong order, are all excellent ways of really plowing up the works.

The second weirdness check concerns the decimal mode. Remember that the 6502 has a hex mode, picked by a CLD command, and a decimal mode, chosen by a SED command. It is always a good idea to reaffirm hex with a CLD as an early program line. If you do use the decimal mode, be sure you get out of it properly, and don't allow any disk access during its use. There's a strange bug called the *$48 problem* that causes disk malfunctions if decimal mode is accidentally gotten into. Another symptom of decimal mode is that scrolling and screen motions behave very erratically.

The third weirdness check is to be sure that variables and registers are not being used for two different things at once by two different points in the program. Apple's own IOSAVE and IOREST are notorious for this sort of thing. Make sure that each subroutine and each interrupt either saves all the registers, or else makes known and safe use of them.

The fourth weirdness check relates to interrupts and interrupting code. If interrupts are allowed, they must be properly initialized and the vectors for their use must be correctly set up. An "early hit" by an interrupt before its vectors are properly set up is bound to mess up everything.

Finally, remember that certain illegal op codes can permanently hang an older 6502 CPU.

So, if you ever have some truly baffling Apple blow-up on your hands, always check these five things early in the game. Chances are one or more of them may be the cause of your grief.

There's only one rule for "stage-two" debugging . . .

"STAGE-TWO" DEBUGGING

Let the code show you the way to go.

You have crossed the viability threshold once your code starts attempting to do something akin to what vaguely resembles more or less approximately what you sort of set out to roughly accomplish in the first place.

At this point, let the code show you the way to go.

Use the code as many different ways as you can, and let it show you what is really needed, and any better ways of going about things. High-level debugging skills are a totally different ball game than the low-level ones. It takes bunches of practice.

Debugging skills take a long time to build. But they are a key process in your becoming a top assembly programmer and going for the brass ring.

SOMETHING OLD, SOMETHING NEW

We are around halfway where I would like to be at this point. I'd really like to get into Apple organization, memory use, peripheral interface, fancy graphics, utility subroutines, superspeed animation, picture processing, and the lots of other exciting new directions we should be going. We'll have to save this stuff for a second volume someday maybe. Use the response card to tell me what you really need or want to see. Or use the support voice hotline that comes free with the companion diskette.

It's almost time to close out the "theory" half of this book, so we can get into the detailed "practice" ripoff modules that remain.

Before we do this, two topics must be mentioned at least in passing. One is very old, and one is very new. These are *Sweet 16* and the *65C02*. . .

SWEET 16—

A very old set of ready-to-use 16-bit "double-wide" software pseudo interpret routines.

65C02—

A brand new, low-power upgrade of the 6502 that includes many more op codes and addressing modes, besides being user extendable.

It never ceases to amaze me who calls what a 16-bit computer. One highly touted personal computer does all its data transfers 8 bits at a time. Yet, they loudly proclaim this to be a 16-bit machine.

On the other hand, Apple has had resident 16-bit utility software available since year one, that immediately and conveniently handles full 16-bit-wide data. Yet, Apple only claims to be an 8-bit machine. So, I guess the difference between an 8-bit and a 16-bit computer depends on who is doing the lying.

Oh well, artistic license and all that.

Anyway, Sweet 16 is a pseudo interpreter code that resides in the old monitor ROM starting at $F689 and gives you sixteen different working registers of sixteen bits each. Use is detailed in the WOZPAK,

in the priceless red book, and in many older club newsletters. You can also steal a copy out of "new" EDASM.

When you activate Sweet 16, it substitutes its own fake op codes that do "full-width" operations for you. When finished, you return back to the plain old 6502 op codes done the regular way. Generally, if you use Sweet 16, your code will be only one third the length, but will also run at one-sixth to one-tenth the speed of your own custom code. Many Apple programs, including EDASM, make use of Sweet 16 internally. A slight reworking or relocation is needed for *Applesloth* compatibility. Three new "long branch" commands have recently been added.

Some assemblers are available that will directly assemble Sweet 16 code for you. The S-C Assembler is one fine example. Ironically, while "old" EDASM uses Sweet 16 internally, it does not recognize or assemble these pseudo-op codes for you. "New" EDASM does both accept and use Sweet 16.

We'll save details on this for another time. But recognize that Sweet 16 gives your Apple some powerful and easy-to-use 16-bit capabilities that save your needing custom code for many uses.

Our "something new" is the 65C02. This, or rather these, are CMOS upgrades of the 6502 microprocessor used in the Apple. Just about every Apple will eventually have a 65C02 as its CPU.

Why?

For one thing, the chip draws far less power. An absolutely cool CPU. It's spooky. In addition, you get bunches of new and powerful addressing modes, as well as lots of extra op codes in existing modes.

For instance, you can now do lots of "pure" indirect commands, without any worry about committing the Y register by forcing it to zero. X and Y can now directly access the stack. You can now test and reset any bit in any position, both page zero and absolute. You now have an unconditional relative branch. You can increment, decrement, or clear the accumulator. Several boo-boos got fixed as well, correcting such things as jump indirect, extra write cycles, and illegal op codes that hang the machine.

The absolutely mind-blowing thing about certain 65C02's is that they are extendable! This means you can add any number of your own op codes and any number of addressing modes that you care to. This is allowable since many of the "illegal" and unused op codes are now *guaranteed* to default to *one-cycle* NOPs. All you do is grab the illegal code and run with it, using some simple external hardware. A 50X speedup in the fastest possible HIRES animation is one of the more mundane possibilities.

In fact, I have a very carefully concocted benchmark here that lets the 65C02 do some very specialized animation-type things much *faster* than a 68000! We've always known that the 6502 was better than the 68000, but now it can be faster as well. Which should drive certain people up the wall.

Unfortunately, at this writing, we are not quite there yet. GTE 65C02's are available and work fine in any Apple, II, II+, or IIe, but lack 32 powerful page zero "bit and branch" commands. The NEC chips won't work in old Apples. Rockwell has those "bit-and-branch" commands, but so far has not guaranteed those all important default NOPs, and, worse yet, has pulled everything back for a redesign. The first Rockwell chips did not run in older Apples.

Mitel doesn't seem to have samples yet. Motorola's delivery of their 65C02 is keyed to "extremely frigid conditions in a rather unpleasant locale," or words to that effect. The Synertek chips appear to be a figment of the catalog writer's imagination. NCR has just begun second sourcing.

All of which should be ancient history by the time you read this. 65C02's should be readily available and extremely useful. Check the ads for availability. The 65C02 is so significant and so powerful that it cannot be ignored by any future-oriented assembly language programmer.

Or by you.

65C02 assembly op codes are now available for the *S-C Assembler,* and should shortly be available for just about all the others. These new op codes are also supported on "new" EDASM.

But, remember that you cannot assemble 65C02 op codes without using an assembler that recognizes them. Also note that most 65C02 code will not run on a stock 6502, and may even hang the machine. Once the bugs are out of the chip, though, all 6502 software should run perfectly on a 65C02. So forward compatibility should be no hassle.

The benefits of the 65C02 are so great, that it is even worth including a free one with your new software. But just about everyone will have one soon enough anyway. The Apple IIc uses a 65C02 as its CPU.

In the wings is a new 16-bit CMOS microprocessor that is, believe it or not, *pin and circuit compatible* with the 6502 and can be dropped into a IIe. Watch for more details on this beauty. Western Design Center is the prime source for this chip.

That should just about wrap up the theory half of whatever it is we are trying to accomplish here. Let me know if you find out.

On to some code that works . . .

MORE DIFFERENCES BETWEEN MY ASSEMBLER AND EDASM:

part II
THE RIPOFF MODULES

HOW TO USE THE RIPOFF
MODULES

The *Ripoff Modules* are a series of nine interactive demos designed to show you how to handle many common Apple machine language programming problems. Each module is listable, completely documented, and out in the open where you can easily access it.

I've tried to emphasize what really gets used, since just about all programming books and most program libraries center on largely outdated, cumbersome, and irrelevant dino stuff, rather than answering the real gut questions, such as "What's the best way to handle lots of text messages that might be mixed and matched together?" "Can I do a fast and well-behaved random number generator?" "Show me how to handle sound effects and musical songs;" or "How can I quickly shuffle cards or rearrange array values?"

Originally, I wanted to have lots of short demo modules. But, there are so many different important things to learn in Apple assembly language that there is simply no way to cram everything into a single book. So, instead, I decided to take the nine things that beginning assembly students seem to have the most trouble with, and expand on these in some depth.

All nine modules and bunches of other goodies are also available on a sanely priced and crammed-full companion diskette, which you can order using the card in the back of this book.

Naturally, full source code is included for each and every module. Of course, the diskette is totally unlocked, unprotected, and fully copyable. You have your choice of EDASM or S-C Assembler formats.

Most other assemblers will accept either of these formats, or else will provide a way to convert them.

A companion voice hotline service goes with this support diskette, similar to the hotline that is provided to *Enhancing* users. This support service is free, except for the usual phone charges.

You are, of course, totally free to adapt and use these ripoff modules in any way you want for any purpose. Just play fair. Give credit for any commercial use and don't try to compete head on.

Most ordinary Apple modules and subroutines are *result* oriented. This means that they are trying to get some job done as quickly and as compactly as possible. Our ripoff modules are *method* oriented instead. I have picked the modules to show you certain ways of handling different programming tasks. I've tried to make each method as mainstream and innovative as possible.

Which means that most of these modules will not have you gripping the edge of your chair in suspense, or rolling in the aisles with laughter over what they are actually doing. The modules are not intended to be arcade-quality entertainment, nor are they supposed to give you spectacular results, when used one at a time by themselves. The modules are intended instead to be a learning tool that shows you how to tackle the real gut issues involved in creating your own Apple machine language programs.

There are lots of ways you can use the ripoff modules . . .

USING THE RIPOFF MODULES
Read about them Run them List them Tear them apart Study them Change them Adapt them Close the loop

Here's how to claim these modules as your own, and to add them to your own programs: First read the background text that goes with each module, so you can see what the module is intended to do. It turns out that any particular programming technique works well for some *range* of complexity, and may be overkill for simpler things and cumbersome or inefficient for very elaborate jobs. So, be sure you understand the intended use of each module, along with any simpler or more complex alternatives.

Next, run the program and watch or listen to it doing its thing. Since many of the modules will stand on their own, what they do will be pretty much limited to pointing out how they work and how they handle a certain task. In some cases, I've "trumped up" a simple example of something complex that the module is supposed to handle. Now, there may be a better way to get the *result* shown by the simple example, but, once again, that's not our point or purpose. We're after *method* here.

Then, reset to the monitor and list the program. Use the "tearing method" from Enhancement 3 of *Enhancing Your Apple II*, Volume I (Sams 21822). Color code each and every disassembled line with a page highlighter, following the *tearing* guidelines. Do this *before* you study the actual source code in depth. The reason is to gain practice reading and understanding machine language listings, particularly for those modules or programs for which you do *not* have source code.

The next step is to compare your "torn" listing against the actual source code shown here in these modules, to be sure you understand exactly what is happening when.

So much for the *analysis*. When you understand the point and purpose of each module, try some *synthesis*.

Capture a copy of the source code for the module, using an assembler of your choice. Then, make some fairly simple changes in the source code, so it will do something "alike but different somehow." Save this to a new diskette, and then assemble and run your new object code. After that, add some bells and whistles to the module's demo so it becomes longer, more interesting, or more exciting.

Now the fun begins. Rewrite the module source code one more time. Only now, make it do something you want it to do in the way you want it done, rather than the way that is shown here. Test your object code, and then actually use it in a larger program of your choosing.

Needless to say, the more time and effort you spend in understanding and capturing these modules, the more value they will be to you.

Finally, close the loop. Use the response card in back or call the hotline to let me know how you have used the existing modules and which new ones you need or would like to see.

Some of the later modules will "borrow" portions of earlier ones to keep the code simple and not reinvent the wheel. We have tried to note what is needed where. The companion diskette also includes an object code program called THE WHOLE BALL OF WAX, which combines all of the ripoff modules together all at once, along with a unifying demo.

Here's a summary of the ripoff modules and what they are intended to do . . .

RIPOFF MODULE SUMMARY

0. THE EMPTY SHELL—

A framework you can use to create most any machine language program of your choosing.

1. FILE BASED PRINTER—

The standard way to output short and fixed text messages using a common message file.

2. IMBEDDED STRING PRINTER—

A much better way to "mix and match" fixed text messages that are imbedded directly into your source code.

3. MONITOR TIME DELAY—

How to use the Apple's WAIT subroutine for animation and other system timing needs.

4. OBNOXIOUS SOUNDS—

A multiple sound effects generator that "calculates" lots of different sounds with minimum code.

5. MUSICAL SONGS—

The standard "red book tones" method of making music, along with a few improvements and upgrades.

6. OPTION PICKER—

How to do menu options or pick modules using the forced subroutine return method.

7. RANDOM NUMBERS—

A fast and usable way to generate "random" numbers without the fatal flaws of the Applesloth "RND" code.

8. SHUFFLE—

An extremely fast "random exchange" method of rearranging an array of numbers or file values.

The ripoff modules each take up one to three pages of memory. Together they sit from hex $6000 through $7300. The location of each module is shown in its source code.

If you want to interact between Applesloth and these modules, just do a HIMEM: 24575 as your first program line. This will protect the module space from being plowed. You can access the code on a PEEK and POKE basis, using your copy of *The Hexadecimal Chronicles* (Sams 21802) to show you the linking points.

One thing we have not, and will not do, is show you BASIC equivalents for the ripoff modules. The whole point of learning assembly language programming is to do so in ways that optimize the use of machine language. Thus, you *never* do something the way BASIC does. That's not even wrong. Only dumb.

On to the modules . . .

0

THE EMPTY SHELL

a framework you can use to create most any machine language program

Here are 500 lines of source code that do—absolutely nothing! It's called the *empty shell* and you use it as a framework for building your own source codes.

Actually, you'll find the empty shell doing lots of good things for you. Since it is usually much easier to edit *existing* code than to *enter* new code on most assemblers, the empty shell makes writing your custom source codes much faster. Secondly, the empty shell forces you to put decent documentation into the program ahead of time, rather than waiting until the last minute and then not doing it. This also keeps your style consistent from program to program.

The empty shell should also give you code that is far cleaner and more understandable. Finally, and most conveniently, the empty shell contains a long machine readable list of practically all of the useful Apple II and IIe subroutines and entry points. Rather than looking these up in a dozen different places, you simply *eliminate* the ones you do *not* want.

The empty shell is, of course, structure, and as we've seen, structure of any sort in a computer program is inherently despicable and *evil*. Nonetheless, we will use the sixteen part structure we looked at back in chapter four.

All the rest of the ripoff modules will show us examples of how the empty shell works and how to use it.

But, where do you start? . . .

> **To use the EMPTY SHELL.SOURCE, first eliminate what you do *not* want or need. This is best done *backward* from end to beginning.**
>
> **Then, edit or change what is left to create your new source code. This is usually done *frontward* from start to finish.**

First, of course, you will want to customize and personalize your own EMPTY SHELL.SOURCE by putting your own name, company, and copyright notice where mine now are. Do not rewrite to the companion diskette. Instead, save everything new on your own new diskettes. That way, you can always return to the originals if disaster strikes.

Here's some more detail on how to go about . . .

USING THE EMPTY SHELL

1. Assemble EMPTY SHELL.SOURCE and make an assembly listing hard copy. Then reload EMPTY SHELL.SOURCE into your assembler or "new way" word processor.

2. Start at the end of the source code and eliminate what you do *not* want. First, check the last line and decide whether you want to use LST OFF or LST ON. LST ON is a good choice for early program versions.

3. Decide how long your program files are to be. If you are using no DFB style files, or if you need less than 256 bytes of single-byte file values, then shorten the DFB section by deleting lines. If you need more file bytes, extend by copying.

4. Go to your hard copy and check off the hooks and constants you are going to use. If you are "old way" editing, put these in alphabetical order and then copy them to the end of their source code listings. Then delete all the unused hooks and constants.

5. Begin editing from the first line. Change the origin, then the title box. Continue editing by rewriting the "What it does," "How to use it," "Gotchas," "Enhancements," and "Random Comments." Don't worry too much about getting these perfect, since you will almost certainly change them as you edit and debug your source code.

6. Add any new hooks and constants that you want to predefine.

7. Enter your high level code and the documentation for the big lumps. Overwrite the NOPs with actual code and comments. Should you need more room, go to the assembler's insert mode and continue.

8. Do the same thing for the little lumps and the crumbs. Then enter your file values.

9. Assemble your new source code and do an assembler listing. Then repair all errors and repeat the process until you get an "error-free" result.

10. Eliminate any spurious lines and comments that may be left over from the original and reassemble.

11. Test your code, and proceed debugging from here just as you would with any other source code.

I've tried to include a fairly complete list of hooks. But note that not every hook will work on every version Apple. For instance, STEP and TRACE will only run with an old autostart ROM, while VBL and ALTCSON will only perform on an Apple IIe.

By the way, I've shortened some of the IIe labels so they are only seven or fewer characters long. You may prefer to use the "official" labels instead.

If you use any "version-specific" Apple features, be sure to include tests in your program to make sure you have the right machine in use. In general, most "mainstream" autostart programs will run on a IIe, but programs that make use of new IIe features will not work on older versions, and may even hang. If you must use some of the "oldies but goodies," it may be best to drag along the needed code *inside* your own program. Stock ID routines are included with "new" EDASM.

Should you be "new way" editing your empty shell, just delete anything unwanted or unneeded as it comes up. Once again, it is best to work from bottom to top in reverse order.

The easiest "old way" means of getting rid of extra and unwanted hooks is to copy those you need to the end of the hook listing and then delete all of the original hooks in one swell foop. With "new way" editing, just chop out what you don't need on the way by.

Since EMPTY SHELL.SOURCE is so complete, it ends up a tad long and rather slow in loading. After you have worked with it for a while, you might like to do a "short form" version of EMPTY SHELL.SOURCE that more meets your specific programming needs. If you do this, keep a printed copy of the original on hand for reference.

A tip . . .

> **ALWAYS do some minor fixup and pretty printing at the same time you make any important source code corrections.**

Whenever you are fixing up fatal errors and making heavy changes in your source code, spend some time to clean up your documentation, improve page breaks, insert spacing, do pretty printing, eliminate unwanted lines, and cosmetic stuff like this. Each reassembly should include both heavy and light repairs.

A good goal is one line of cosmetic fix for each line of heavy fix.

This way, by the time you finally get your program debugged and working, it also will be pretty much properly documented and attractive to look at. Whatever you do, don't save the documentation for last. Start with your documentation. Sharpen, improve, clarify as you go along.

All the rest of the ripoff modules were written using the EMPTY SHELL.SOURCE. Use these as study examples, and then work up your own custom shell that meets your personal programming needs.

MIND BENDERS

— Write a WPL program that automates your empty shell setup, through use of prompts and directed questions.

— If your "new way" word processor has glossary or user-defined keys, show how to use these for single key macros and other speedup tricks.

— Solve the new way tabbing problem so that active source code lines position themselves correctly, yet comment lines remain intact.

— What tests should your source code include to make sure of . . .

    —uppercase vs lowercase?
    —II vs IIe?
    —40 vs 80 column?
    —paddles vs joystick?
    —joystick orientation?

PROGRAM RM-0
THE EMPTY SHELL

```
----- NEXT OBJECT FILE NAME IS EMPTY SHELL
6000:              3              ORG  $6000      ; ORIGIN GOES HERE


6000:              5 ;   *******************************************
6000:              6 ;   *                                         *
6000:              7 ;   *           -< THE   EMPTY   SHELL >-      *
6000:              8 ;   *                                         *
6000:              9 ;   *              (DUMMY PROGRAM)            *
6000:             10 ;   *                                         *
6000:             11 ;   *        VERSION 1.0   ($6000-$6160)     *
6000:             12 ;   *                                         *
6000:             13 ;   *                 5-24-83                 *
6000:             14 ;   *.......................................*
6000:             15 ;   *                                         *
6000:             16 ;   *           COPYRIGHT C 1983 BY          *
6000:             17 ;   *                                         *
6000:             18 ;   *       DON LANCASTER AND SYNERGETICS     *
6000:             19 ;   *       BOX 1300, THATCHER AZ., 85552    *
6000:             20 ;   *                                         *
6000:             21 ;   *       ALL COMMERCIAL RIGHTS RESERVED    *
6000:             22 ;   *                                         *
6000:             23 ;   *******************************************


6000:             25 ;             *** WHAT IT DOES ***

6000:             27 ;   THIS PROGRAM IS A DUMMY SHELL USED AS A STARTING
6000:             28 ;   POINT FOR YOUR OWN ASSEMBLY LANGUAGE PROGRAMS.
6000:             29 ;
6000:             30 ;
6000:             31 ;
6000:             32 ;


6000:             34 ;             *** HOW TO USE IT ***

6000:             36 ;   TO USE, EDIT THE PROGRAM BY MOVING THE ORIGIN,
6000:             37 ;   CHANGING THE TITLE, REMOVING EXTRA EQU'S, ADDING
6000:             38 ;   YOUR OWN WORKING CODE, ALTERING THE DATA FILES
6000:             39 ;   AND DOING WHATEVER ELSE MAY BE NEEDED TO BUILD
6000:             40 ;   YOUR OWN CUSTOM ASSEMBLED PROGRAM OR MODULE.
6000:             41 ;
```

PROGRAM RM-0, CONT'D . . .

```
6000:          44 ;              *** GOTCHAS ***

6000:          46 ;    ANYTHING ESSENTIAL FOR USE OR UNDERSTANDING OF THE
6000:          47 ;    PROGRAM GETS PUT HERE.  THIS INCLUDES SPECIAL NEEDS
6000:          48 ;    SUCH AS EXTRA MEMORY, ANY COMPANION CODE MODULES, OR
6000:          49 ;    ANY SPECIAL HARDWARE.
6000:          50 ;
6000:          51 ;



6000:          53 ;             *** ENHANCEMENTS ***

6000:          55 ;    PUT ANY ADD-ONS, "EXTRA TRICKS", OR SPECIAL
6000:          56 ;    USES HERE.  INCLUDE USE TIPS AND APPLICATIONS.
6000:          57 ;
6000:          58 ;
6000:          59 ;
6000:          60 ;



6000:          62 ;            *** RANDOM COMMENTS ***

6000:          64 ;    IF THERE IS SOMETHING ELSE YOU WANT TO SAY THAT'S
6000:          65 ;    NOT ALL THAT IMPORTANT, YOU CAN ADD IT IN THIS SPACE.
6000:          66 ;
6000:          67 ;
6000:          68 ;
6000:          69 ;
```

PROGRAM RM-0, CONT'D . . .

```
6000:           72 ;           *** HOOKS ***


0020:           74 WNDLFT   EQU  $20        ; SCROLL WINDOW LEFT
0021:           75 WNDWDTH  EQU  $21        ; SCROLL WINDOW WIDTH
0022:           76 WNDTOP   EQU  $22        ; SCROLL WINDOW TOP
0023:           77 WNDBOT   EQU  $23        ; SCROLL WINDOW BOTTOM
0024:           78 CH       EQU  $24        ; CURSOR HORIZONTAL
0025:           79 CV       EQU  $25        ; CURSOR VERTICAL
0026:           80 GBASL    EQU  $26        ; LORES BASE LOW
0027:           81 GBASH    EQU  $27        ; LORES BASE HIGH
0028:           82 BASL     EQU  $28        ; TEXT BASE LOW
0029:           83 BASH     EQU  $29        ; TEXT BASE HIGH
002C:           84 HEND     EQU  $2C        ; LORES RIGHT END H LINE
002D:           85 VBOT     EQU  $2D        ; LORES BOTTOM OF V LINE
0030:           86 COLOR    EQU  $30        ; LORES COLOR
0031:           87 INVFLG   EQU  $31        ; NORMAL/INVERSE/FLASH (FF,7F,3F)
0033:           88 PROMPT   EQU  $33        ; HOLDS PROMPT SYMBOL
0036:           89 CSWL     EQU  $36        ; OUTPUT CHARACTER HOOK LOW
0037:           90 CSWH     EQU  $37        ; OUTPUT CHARACTER HOOK HIGH
0038:           91 KSWL     EQU  $38        ; INPUT CHARACTER HOOOK LOW
0039:           92 KSWH     EQU  $39        ; INPUT CHARACTER HOOK HIGH
004E:           93 RNDL     EQU  $4E        ; RANDOM NUMBER LOW
004F:           94 RNDH     EQU  $4F        ; RANDOM NUMBER HIGH


0100:           96 STACK    EQU  $0100      ; STACK PAGE ACCESS


0200:           98 KEYBUF   EQU  $0200      ; KEYBUFFER START


03D0:          100 DOSWRM   EQU  $03D0      ; DOS WARM START JMP
03D3:          101 DOSCLD   EQU  $03D3      ; DOS COLD START JMP
03D6:          102 DOSFLM   EQU  $03D6      ; DOS FILE MANAGER JUMP
03D9:          103 DOSRWTS  EQU  $03D9      ; DOS RWTS JUMP
03DC:          104 DOSIPRM  EQU  $03DC      ; DOS FILE PARAMETER FIND JUMP
03E3:          105 DOSRWLS  EQU  $03E3      ; DOS RWTS PARAMETER FIND JUMP
03EA:          106 DOSHOOK  EQU  $03EA      ; DOS HOOK RECONNECT JUMP
03F0:          107 BRK      EQU  $03F0      ; BREAK ADDRESS (AUTOSTART& 2E ONLY!)
03F2:          108 SOFTEV   EQU  $03F2      ; SOFT RESET (AUTOSTART & 2E ONLY!)
03F4:          109 PWRDUP   EQU  $03F4      ; WARM START EOR CHECKSUM
03F5:          110 AMPERV   EQU  $03F5      ; APPLESOFT "&" JUMP
03F8:          111 USRADR   EQU  $03F8      ; CTRL-Y JUMP
03FB:          112 NMI      EQU  $03FB      ; NON-MASKABLE INTERRUPT JUMP
03FE:          113 IRQLOC   EQU  $03FE      ; INTERRUPT ADDRESS LOW


0400:          115 TEXTP1   EQU  $0400      ; START OF TEXT PAGE ONE
0800:          116 TEXTP2   EQU  $0800      ; START OF TEXT PAGE TWO
2000:          117 HIRESP1  EQU  $2000      ; START OF HIRES PAGE ONE
4000:          118 HIRESP2  EQU  $4000      ; START OF HIRES PAGE TWO
```

PROGRAM RM-0, CONT'D . . .

```
C000:     121 IOADR    EQU  $C000    ; KEYBOARD INPUT
C010:     122 KBDSTR   EQU  $C010    ; KEYSTROBE RESET
C020:     123 TAPEOUT  EQU  $C020    ; CASSETTE OR AUDIO OUT
C030:     124 SPKR     EQU  $C030    ; SPEAKER CLICK OUTPUT
C040:     125 STROBE   EQU  $C040    ; GAME CONNECTOR STROBE
C050:     126 TXTCLR   EQU  $C050    ; GRAPHICS ON SOFT SWITCH
C051:     127 TXTSET   EQU  $C051    ; TEXT ON SOFT SWITCH
C052:     128 MIXCLR   EQU  $C052    ; FULL SCREEN SOFT SWITCH
C053:     129 MIXSET   EQU  $C053    ; MIXED SCREEN SOFT SWITCH
C054:     130 LOWSCR   EQU  $C054    ; PAGE ONE SOFT SWITCH
C055:     131 HISCR    EQU  $C055    ; PAGE TWO SOFT SWITCH
C056:     132 LORES    EQU  $C056    ; LORES SOFT SWITCH
C057:     133 HIRES    EQU  $C057    ; HIRES SOFT SWITCH
C060:     134 PB4      EQU  $C060    ; CASS IN + "FOURTH" PB INPUT "SW3"
C061:     135 PB1      EQU  $C061    ; OPEN APPLE + "FIRST" PB INPUT "SW0"
C062:     136 PB2      EQU  $C062    ; CLOSED APPLE + "SECOND" PB INPUT "S
C063:     137 PB3      EQU  $C063    ; "THIRD"  PUSHBUTTON INPUT "SW2"
C064:     138 PDL0     EQU  $C064    ; GAME PADDLE 0 ANALOG IN
C065:     139 PDL1     EQU  $C065    ; GAME PADDLE 1 ANALOG IN
C066:     140 PDL2     EQU  $C066    ; GAME PADDLE 2 ANALOG IN
C067:     141 PDL3     EQU  $C067    ; GAME PADDLE 3 ANALOG IN
C070:     142 PTRIG    EQU  $C070    ; ANALOG PADDLE RESET


C080:     144 STEP00   EQU  $C080    ; DISK STEPPER PHASE 0 OFF
C081:     145 STEP01   EQU  $C081    ; DISK STEPPER PHASE 0 ON
C082:     146 STEP10   EQU  $C082    ; DISK STEPPER PHASE 1 OFF
C083:     147 STEP11   EQU  $C083    ; DISK STEPPER PHASE 1 ON
C084:     148 STEP20   EQU  $C084    ; DISK STEPPER PHASE 2 OFF
000C:     149 STEP21   EQU  $CO85    ; DISK STEPPER PHASE 2 ON
C086:     150 STEP30   EQU  $C086    ; DISK STEPPER PHASE 3 OFF
C087:     151 STEP31   EQU  $C087    ; DISK STEPPER PHASE 3 ON
C088:     152 MOTON    EQU  $C088    ; DISK MAIN MOTOR OFF
C089:     153 MOTOFF   EQU  $C089    ; DISK MAIN MOTOR ON
C08A:     154 DRV0EN   EQU  $C08A    ; DISK ENABLE DRIVE 1
C08B:     155 DRV1EN   EQU  $C08B    ; DISK ENABLE DRIVE 2
C08C:     156 Q6CLR    EQU  $C08C    ; DISK Q6 CLEAR
C08D:     157 Q6SET    EQU  $C08D    ; DISK Q6 SET
C08E:     158 Q7CLR    EQU  $C08E    ; DISK Q7 CLEAR
C08F:     159 Q7SET    EQU  $C08F    ; DISK Q7 SET


E000:     161 BASICLD  EQU  $E000    ; ENTER BASIC COLD
E003:     162 BASICWM  EQU  $E003    ; RE-ENTER BASIC WARM


F3D8:     164 HGR2     EQU  $F3D8    ; APPLESOFT CLEAR TO HIRES 2
F3E2:     165 HGR      EQU  $F3E2    ; APPLESOFT CLEAR TO HIRES 1
F3F4:     166 BKGND    EQU  $F3F4    ; APPLESOFT HIRES BACKGROUND CLEAR
F6F0:     167 HCOLOR   EQU  $F6F0    ; APPLESOFT HIRES COLOR SELECT
F411:     168 HPOSN    EQU  $F411    ; APPLESOFT HIRES POSITION
F457:     169 HPLOT    EQU  $F457    ; APPLESOFT HIRES PLOT
```

PROGRAM RM-0, CONT'D . . .

```
F800:        172 PLOT    EQU  $F800    ; PLOT LORES BLOCK
F819:        173 HLINE   EQU  $F819    ; HORIZ LORES LINE
F828:        174 VLINE   EQU  $F828    ; VERTICAL LORES LINE
F832:        175 CLRSCR  EQU  $F832    ; CLEAR FULL LORES SCREEN
F836:        176 CLRTOP  EQU  $F836    ; CLEAR TOP LORES SCREEN
F847:        177 GBSCALC EQU  $F847    ; LORES BASE CALCULATION
F85F:        178 NEXTCOL EQU  $F85F    ; INCREASE LORES COLOR BY 3
F864:        179 SETCOL  EQU  $F864    ; SET LORES COLOR
F871:        180 SCRN    EQU  $F871    ; READ LORES SCREEN COLOR
F941:        181 PRNTAX  EQU  $F941    ; OUTPUT A THEN X AS HEX
F948:        182 PRBLNK  EQU  $F948    ; OUTPUT 3 SPACES VIA HOOKS
F94A:        183 PRBL2   EQU  $F94A    ; OUTPUT X BLANKS VIA HOOKS


FAD7:        185 REGDSP  EQU  $FAD7    ; DISPLAY WORKING REGISTERS
FB1E:        186 PREAD   EQU  $FB1E    ; READ GAME PADDLE X
FB2F:        187 INIT    EQU  $FB2F    ; INITIALIZE TEXT SCREEN
FB93:        188 SETTXT  EQU  $FB93    ; SET UP TEXT SCREEN (NOT 2E!)
FB40:        189 SETGR   EQU  $FB40    ; SET UP GRAPHICS SCREEN
FB4B:        190 SETWND  EQU  $FB4B    ; SET NORMAL TEXT WINDOW
FBC1:        191 BASCALC EQU  $FBC1    ; CALCULATE TEXT BASE ADDRESS (NOT 2E!)
FBD9:        192 BELL1   EQU  $FBD9    ; BEEP SPEAKER IF CTRL-G
FBE4:        193 BELL2   EQU  $FBE4    ; BEEP SPEAKER ONCE
FBF4:        194 ADVANCE EQU  $FBF4    ; TEXT CURSOR ONE TO RIGHT
FBFD:        195 VIDOUT  EQU  $FBFD    ; OUTPUT ASCII TO SCREEN ONLY


FC10:        197 BS      EQU  $FC10    ; BACKSPACE SCREEN
FC1A:        198 UP      EQU  $FC1A    ; MOVE SCREEN CURSOR UP ONE LINE
FC22:        199 VTAB    EQU  $FC22    ; VERTICAL SCREEN TAB USING CV
FC24:        200 VTABA   EQU  $FC24    ; VERTICAL SCREEN TAB USING A
FC66:        201 ESC1    EQU  $FC66    ; PROCESS ESCAPE CURSOR MOVES
FC42:        202 CLREOP  EQU  $FC42    ; CLEAR TO END OF PAGE
FC58:        203 HOME    EQU  $FC58    ; CLEAR TEXT SCREEN AND HOME CURSOR
FC62:        204 CR      EQU  $FC62    ; CARRIAGE RETURN TO SCREEN
FC66:        205 LF      EQU  $FC66    ; LINEFEED TO SCREEN ONLY
FC70:        206 SCROLL  EQU  $FC70    ; SCROLL TEXT SCREEN UP ONE
FC9C:        207 CLEOL   EQU  $FC9C    ; CLEAR TEXT TO END OF LINE
FCA8:        208 WAIT    EQU  $FCA8    ; TIME DELAY SET BY ACCUMULATOR
FD0C:        209 RDKEY   EQU  $FD0C    ; GET INPUT CHARACTER VIA HOOKS
FD1B:        210 KEYIN   EQU  $FD1B    ; READ THE APPLE KEYBOARD
FD35:        211 RDCHAR  EQU  $FD35    ; GET KEY AND PROCESS ESC A-F
FD62:        212 CANCEL  EQU  $FD62    ; CANCEL KEYBOARD LINE ENTRY
FD67:        213 GETLNZ  EQU  $FD67    ; CR THEN GET KEYBOARD INPUT LINE
FD6A:        214 GETLN   EQU  $FD6A    ; GET KEYBOARD INPUT LINE
FD6F:        215 GETLN1  EQU  $FD6F    ; GET KBD INPUT, NO PROMPT
FD8B:        216 CROUT1  EQU  $FD8B    ; CLEAR EOL THEN CR VIA HOOKS
FD8E:        217 CROUT   EQU  $FD8E    ; OUTPUT CR VIA HOOKS
FDDA:        218 PRBYTE  EQU  $FDDA    ; OUTPUT FULL A IN HEX TO HOOKS
FDE3:        219 PRHEX   EQU  $FDE3    ; OUTPUT LOW A IN HEX TO HOOKS
FDED:        220 COUT    EQU  $FDED    ; OUTPUT CHARACTER VIA HOOKS
FDF0:        221 COUT1   EQU  $FDF0    ; OUTPUT CHARACTER TO SCREEN
```

PROGRAM RM-0, CONT'D . . .

```
FE2C:      224 MOVE    EQU  $FE2C    ; MOVE BLOCK OF MEMORY
FE36:      225 VERIFY  EQU  $FE36    ; VERIFY BLOCK OF MEMORY
FE5E:      226 LIST    EQU  $FE5E    ; DISASSEMBLE 20 INSTRUCTIONS
FE63:      227 LIST2   EQU  $FE63    ; DISASSEMBLE "A" INSTRUCTIONS
FE80:      228 SETINV  EQU  $FE80    ; PRINT INVERSE TEXT TO SCREEN
FE84:      229 SETNORM EQU  $FE84    ; PRINT NORMAL TEXT TO SCREEN
FE93:      230 SETVID  EQU  $FE93    ; GRAB OUTPUT HOOKS FOR SCREEN
FEB0:      231 XBASIC  EQU  $FEB0    ; GO BASIC, DESTROYING OLD
FEB3:      232 BASCON  EQU  $FEB3    ; GO BASIC, CONTINUING OLD
FEC2:      233 TRACE   EQU  $FEC2    ; START TRACING (OLD ROM ONLY!)
FEC4:      234 STEP    EQU  $FEC4    ; SINGLE STEP (OLD ROM ONLY!)
FECD:      235 WRITE   EQU  $FECD    ; WRITE TO CASSETTE TAPE
FEFD:      236 READ    EQU  $FEFD    ; READ FROM CASSETTE TAPE
FF2D:      237 PRERR   EQU  $FF2D    ; PRINT "ERR" TO OUTPUT HOOK
FF3A:      238 BELL    EQU  $FF3A    ; OUTPUT BELL TO HOOKS
FF3F:      239 IORESR  EQU  $FF3F    ; RESTORE ALL WORKING REGISTERS
FF4A:      240 IOSAVE  EQU  $FF4A    ; SAVE ALL WORKING REGISTERS
FF58:      241 RETURN  EQU  $FF58    ; "GUARANTEED" RETURN
FF59:      242 OLDRST  EQU  $FF59    ; OLD RESET, NO AUTOSTART
FF65:      243 MON     EQU  $FF65    ; ENTER MONITOR AND BEEP SPEAKER
FF69:      244 MONZ    EQU  $FF69    ; ENTER MONITOR QUIETLY
FFA7:      245 GETNUM  EQU  $FFA7    ; ASCII TO HEX IN 3E & 3F


6000:      247 ;                *** HOOKS FOR 2E ONLY! ***


C000:      249 CLR80CO EQU  $C000    ; 80 STORE OFF (WRITE ONLY)
C001:      250 SET80CO EQU  $C001    ; 80 STORE ON  (WRITE ONLY)
C002:      251 RAMRDMN EQU  $C002    ; READ MAIN MEMORY (WRITE ONLY)
C003:      252 RAMRDAX EQU  $C003    ; READ AUXILIARY MEMORY (WRITE ONLY)
C004:      253 RAMWRMN EQU  $C004    ; WRITE MAIN MEMORY (WRITE ONLY)
C005:      254 RAMWRAX EQU  $C005    ; WRITE AUXILIARY MEMORY (WRITE ONLY)
C006:      255 SLOTXRM EQU  $C006    ; INTERNAL ROM AT CX00 (WRITE ONLY)
C007:      256 SLOTXEX EQU  $C007    ; SLOT ROM AT CX00 (WRITE ONLY)


C008:      258 MAINZP  EQU  $C008    ; USE MAIN ZERO PAGE (WRITE ONLY)
C009:      259 ALTZP   EQU  $C009    ; USE ALTERNATE ZERO PAGE (WRITE ONLY)
C00A:      260 SLOT3RM EQU  $C00A    ; SLOT #3 INTERNAL ROM (WRITE ONLY)
C00B:      261 SLOT3EX EQU  $C00B    ; SLOT #3 EXTERNAL ROM (WRITE ONLY)
C00C:      262 OFF80CL EQU  $C00C    ; TURN 80 COLUMN OFF (WRITE ONLY)
C00D:      263 ON80COL EQU  $C00D    ; TURN 80 COLUMN ON (WRITE ONLY)
C00E:      264 ALTCSOF EQU  $C00E    ; USE MAIN CHARACTER SET (WRITE ONLY)
C00F:      265 ALTCSON EQU  $C00F    ; USE ALT CHARACTER SET (WRITE ONLY)
```

PROGRAM RM-0, CONT'D . . .

```
C013:          268 RAMRDS  EQU  $C013   ; READ RAMREAD SWITCH (READ ONLY)
C014:          269 RAMWTS  EQU  $C014   ; READ RAMWRITE SWITCH (READ ONLY)
C015:          270 SLTCXS  EQU  $C015   ; READ SLOT CX SWITCH (READ ONLY)
C016:          271 ALTZPS  EQU  $C016   ; READ ZERO PAGE SWITCH (READ ONLY)
C017:          272 SLTC3S  EQU  $C017   ; READ SLOT C3 SWITCH (READ ONLY)


C018:          274 S80STR  EQU  $C018   ; READ 80STORE SWITCH (READ ONLY)
C019:          275 VBL     EQU  $C019   ; VERT. BLANKING >80=BLANK (READ ONLY)
C01A:          276 TEXTS   EQU  $C01A   ; READ TEXT SWITCH (READ ONLY)
C01B:          277 MIXEDS  EQU  $C01B   ; READ MIXED GR SWITCH (READ ONLY)
C01C:          278 PAGE2S  EQU  $C01C   ; READ PAGE 2 SWITCH (READ ONLY)
C01D:          279 HIRESS  EQU  $C01D   ; READ HIRES SWITCH (READ ONLY)
C01E:          280 ALTCSS  EQU  $C01E   ; READ ALTCHAR SET SWITCH (READ ONLY)
C01F:          281 S80COL  EQU  $C01F   ; READ 80 COLUMN SWITCH (READ ONLY)


C080:          283 RB2RAM  EQU  $C080   ; READ BANK 2 RAM
C081:          284 WB2RAM  EQU  $C081   ; WRITE BANK 2 RAM, READ ROM
C082:          285 RROM    EQU  $C082   ; READ ROM ONLY, NO WRITE
C083:          286 RWRAM2  EQU  $C083   ; READ & WRITE RAM2 (HIT TWICE!)
C088:          287 RRAM1   EQU  $C088   ; READ BANK1 RAM
C089:          288 WRAM1   EQU  $C089   ; WRITE BANK1 RAM, READ ROM
C08A:          289 RB1ROM  EQU  $C08A   ; READ BANK1 ROM
C08B:          290 RWRAM1  EQU  $C08B   ; READ & WRITE RAM1 (HIT TWICE!)


6000:          292 ;              *** CONSTANTS ***
6000:          293 ;         *** TEXTFILE COMMANDS ***


0088:          295 B       EQU  $88     ; BACKSPACE
008D:          296 C       EQU  $8D     ; CARRIAGE RETURN
0084:          297 D       EQU  $84     ; DOS ATTENTION
008C:          298 F       EQU  $8C     ; FORMFEED
0087:          299 G       EQU  $87     ; RING GONG
008A:          300 L       EQU  $8A     ; LINEFEED
0060:          301 P       EQU  $60     ; FLASHING PROMPT
0000:          302 X       EQU  $00     ; END OF MESSAGE
```

PROGRAM RM-0, CONT'D . . .

```
6000:           305 ;               *** BIG LUMPS ***
6000:           306 ;              *** MAIN PROGRAM ***
6000:           307 ;            *** HIGH LEVEL CODE ***


6000:           309 ;     ADD ANY COMMENTS HERE THAT ARE
6000:           310 ;     SPECIFIC TO THE BIG LUMPS.
6000:           311 ;
6000:           312 ;
6000:           313 ;
6000:           314 ;


6000:EA         316 START1  NOP          ; YOUR HIGH LEVEL CODE STARTS HERE
6001:EA         317         NOP          ;   AND GOES ON AS FAR AS NEEDED.
6002:EA         318         NOP          ;
6003:EA         319         NOP          ;
6004:EA         320         NOP          ;
6005:EA         321         NOP          ;
6006:EA         322         NOP          ;
6007:EA         323         NOP          ;

6008:EA         325         NOP          ;
6009:EA         326         NOP          ;
600A:EA         327         NOP          ;
600B:EA         328         NOP          ;
600C:EA         329         NOP          ;
600D:EA         330         NOP          ;
600E:EA         331         NOP          ;
600F:EA         332         NOP          ;

6010:EA         334         NOP          ;
6011:EA         335         NOP          ;
6012:EA         336         NOP          ;
6013:EA         337         NOP          ;
6014:EA         338         NOP          ;
6015:EA         339         NOP          ;
6016:EA         340         NOP          ;
6017:EA         341         NOP          ;

6018:EA         343         NOP          ;
6019:EA         344         NOP          ;
601A:EA         345         NOP          ;
601B:EA         346         NOP          ;
601C:EA         347         NOP          ;
601D:EA         348         NOP          ;
601E:EA         349         NOP          ;
601F:EA         350         NOP          ;
```

PROGRAM RM-0, CONT'D . . .

```
6020:           353 ;            *** LITTLE  LUMPS ***
6020:           354 ;          *** HEAVY SUBROUTINE ***
6020:           355 ;          *** SUPPORTING  MODULE ***


6020:·          357 ;          ADD ANY COMMENTS HERE THAT ARE
6020:·          358 ;          SPECIFIC TO THE LITTLE LUMPS.
6020:           359 ;
6020:           360 ;
6020:           361 ;
6020:           362 ;


6020:EA         364 START2  NOP           ; YOUR MEDIUM LEVEL CODE STARTS
6021:EA         365         NOP           ;  HERE AND GOES ON AS FAR AS
6022:EA         366         NOP           ;  NEEDED.
6023:EA         367         NOP           ;
6024:EA         368         NOP           ;
6025:EA         369         NOP           ;
6026:EA         370         NOP           ;
6027:EA         371         NOP           ;

6028:EA         373         NOP           ;
6029:EA         374         NOP           ;
602A:EA         375         NOP           ;
602B:EA         376         NOP           ;
602C:EA         377         NOP           ;
602D:EA         378         NOP           ;
602E:EA         379         NOP           ;
602F:EA         380         NOP           ;

6030:EA         382         NOP           ;
6031:EA         383         NOP           ;
6032:EA         384         NOP           ;
6033:EA         385         NOP           ;
6034:EA         386         NOP           ;
6035:EA         387         NOP           ;
6036:EA         388         NOP           ;
6037:EA         389         NOP           ;

6038:EA         391         NOP           ;
6039:EA         392         NOP           ;
603A:EA         393         NOP           ;
603B:EA         394         NOP           ;
603C:EA         395         NOP           ;
603D:EA         396         NOP           ;
603E:EA         397         NOP           ;
603F:EA         398         NOP           ;
```

PROGRAM RM-0, CONT'D . . .

```
6040:        401 ;              ***  STASH   ***
6040:        402 ;             **** THE CRUMBS ***
6040:        403 ;             *** DETAIL  SUBS ***


6040:        405 ;         ADD ANY COMMENTS HERE THAT
6040:        406 ;         ARE SPECIFIC TO THE CRUMBS.
6040:        407 ;
6040:        408 ;
6040:        409 ;
6040:        410 ;


6040:EA      412 START3   NOP          ; YOUR LOW LEVEL CODE STARTS HERE AND
6041:EA      413          NOP          ;  INCLUDES ANY SHORT FILES THAT ARE
6042:EA      414          NOP          ;  RARELY CHANGED.
6043:EA      415          NOP          ;
6044:EA      416          NOP          ;
6045:EA      417          NOP          ;
6046:EA      418          NOP          ;
6047:EA      419          NOP          ;

6048:EA      421          NOP          ;
6049:EA      422          NOP          ;
604A:EA      423          NOP          ;
604B:EA      424          NOP          ;
604C:EA      425          NOP          ;
604D:EA      426          NOP          ;
604E:EA      427          NOP          ;
604F:EA      428          NOP          ;

6050:EA      430          NOP          ;
6051:EA      431          NOP          ;
6052:EA      432          NOP          ;
6053:EA      433          NOP          ;
6054:EA      434          NOP          ;
6055:EA      435          NOP          ;
6056:EA      436          NOP          ;
6057:EA      437          NOP          ;

6058:EA      439          NOP          ;
6059:EA      440          NOP          ;
605A:EA      441          NOP          ;
605B:EA      442          NOP          ;
605C:EA      443          NOP          ;
605D:EA      444          NOP          ;
605E:EA      445          NOP          ;
605F:EA      446          NOP          ;
```

PROGRAM RM-0, CONT'D . . .

```
6060:               449 ;               *** MAIN FILES ***


6060:               451 ;       ADD ANY COMMENTS HERE THAT ARE
6060:               452 ;       SPECIFIC TO THE MAIN FILES.
6060:               453 ;
6060:               454 ;
6060:               455 ;
6060:               456 ;


6060:00 00 00       458 FILE1    DFB        $00,$00,$00,$00,$00,$00,$00,$00
6063:00 00 00
6066:00 00
6068:00 00 00       459 FILE2    DFB        $00,$00,$00,$00,$00,$00,$00,$00
606B:00 00 00
606E:00 00
6070:00 00 00       460 FILE3    DFB        $00,$00,$00,$00,$00,$00,$00,$00
6073:00 00 00
6076:00 00
6078:00 00 00       461 FILE4    DFB        $00,$00,$00,$00,$00,$00,$00,$00
607B:00 00 00
607E:00 00
6080:00 00 00       462 FILE5    DFB        $00,$00,$00,$00,$00,$00,$00,$00
6083:00 00 00
6086:00 00
6088:00 00 00       463 FILE6    DFB        $00,$00,$00,$00,$00,$00,$00,$00
608B:00 00 00
608E:00 00
6090:00 00 00       464 FILE7    DFB        $00,$00,$00,$00,$00,$00,$00,$00
6093:00 00 00
6096:00 00
6098:00 00 00       465 FILE8    DFB        $00,$00,$00,$00,$00,$00,$00,$00
609B:00 00 00
609E:00 00
60A0:00 00 00       466 FILE9    DFB        $00,$00,$00,$00,$00,$00,$00,$00
60A3:00 00 00
60A6:00 00
60A8:00 00 00       467 FILE10   DFB        $00,$00,$00,$00,$00,$00,$00,$00
60AB:00 00 00
60AE:00 00
60B0:00 00 00       468 FILE11   DFB        $00,$00,$00,$00,$00,$00,$00,$00
60B3:00 00 00
60B6:00 00
60B8:00 00 00       469 FILE12   DFB        $00,$00,$00,$00,$00,$00,$00,$00
60BB:00 00 00
60BE:00 00
60C0:00 00 00       470 FILE13   DFB        $00,$00,$00,$00,$00,$00,$00,$00
60C3:00 00 00
60C6:00 00
60C8:00 00 0C       471 FILE14   DFB        $00,$00,$00,$00,$00,$00,$00,$00
60CB:00 00 00
60CE:00 00
```

PROGRAM RM-0, CONT'D . . .

```
60D0:00 00 00    474 FILE15   DFB         $00,$00,$00,$00,$00,$00,$00,$00
60D3:00 00 00
60D6:00 00
60D8:00 00 00    475 FILE16   DFB         $00,$00,$00,$00,$00,$00,$00,$00
60DB:00 00 00
60DE:00 00
60E0:00 00 00    476 FILE17   DFB         $00,$00,$00,$00,$00,$00,$00,$00
60E3:00 00 00
60E6:00 00
60E8:00 00 00    477 FILE18   DFB         $00,$00,$00,$00,$00,$00,$00,$00
60EB:00 00 00
60EE:00 00
60F0:00 00 00    478 FILE19   DFB         $00,$00,$00,$00,$00,$00,$00,$00
60F3:00 00 00
60F6:00 00
60F8:00 00 00    479 FILE20   DFB         $00,$00,$00,$00,$00,$00,$00,$00
60FB:00 00 00
60FE:00 00
6100:00 00 00    480 FILE21   DFB         $00,$00,$00,$00,$00,$00,$00,$00
6103:00 00 00
6106:00 00
6108:00 00 00    481 FILE22   DFB         $00,$00,$00,$00,$00,$00,$00,$00
610B:00 00 00
610E:00 00
6110:00 00 00    482 FILE23   DFB         $00,$00,$00,$00,$00,$00,$00,$00
6113:00 00 00
6116:00 00
6118:00 00 00    483 FILE24   DFB         $00,$00,$00,$00,$00,$00,$00,$00
611B:00 00 00
611E:00 00
6120:00 00 00    484 FILE25   DFB         $00,$00,$00,$00,$00,$00,$00,$00
6123:00 00 00
6126:00 00
6128:00 00 00    485 FILE26   DFB         $00,$00,$00,$00,$00,$00,$00,$00
612B:00 00 00
612E:00 00
6130:00 00 00    486 FILE27   DFB         $00,$00,$00,$00,$00,$00,$00,$00
6133:00 00 00
6136:00 00
6138:00 00 00    487 FILE28   DFB         $00,$00,$00,$00,$00,$00,$00,$00
613B:00 00 00
613E:00 00
6140:00 00 00    488 FILE29   DFB         $00,$00,$00,$00,$00,$00,$00,$00
6143:00 00 00
6146:00 00
6148:00 00 00    489 FILE30   DFB         $00,$00,$00,$00,$00,$00,$00,$00
614B:00 00 00
614E:00 00
6150:00 00 00    490 FILE31   DFB         $00,$00,$00,$00,$00,$00,$00,$00
6153:00 00 00
6156:00 00
6158:00 00 00    491 FILE32   DFB         $00,$00,$00,$00,$00,$00,$00,$00
615B:00 00 00
615E:00 00
```

PROGRAM RM-0, CONT'D . . .

```
6160:          494 ;          *** BOTTOM LINE COMMENTS ***


6160:          496 ;     ADD ANY FINAL COMMENTS YOU FEEL
6160:          497 ;     ARE NEEDED IN THIS SPACE.



*** SUCCESSFUL ASSEMBLY: NO ERRORS
```

FILE BASED PRINTER

**the "standard" way to output
short and fixed text messages
by using a common message
file.**

Outputting text is probably the most fundamental and most impor-
tant task we would ever ask of a machine language Apple program.
You might want to use the text to create a printed record, to inform
the user via the video screen, or to pass a command to the disk
system.

It turns out that there is no "best" way to go about outputting text
from machine language. Instead, there are many different methods
you can pick. These methods are based on *how many* messages you
must output, on *how long* each message is, and on *how changeable*
the messages have to be.

Further, you have to decide just *where* your message is going to go
as well. Usually, to output a character, you get it from somewhere and
put it in the accumulator. Then you go to a text outputting subroutine
that puts the character where you want it to appear. You continue this
until some change occurs, such as a marker or length count. Then you
go on to the next task at hand.

Here are some possible . . .

PLACES TO OUTPUT TEXT

Direct store to the text screen
To COUT hook subroutine $FDF0
To COUT1 screen subroutine $FDED
To a HIRES character generator
To your own custom code

If you *direct store* to a screen location, you end up putting characters on the screen in the shortest possible time, and you are always sure *exactly* where on the screen the character is to go. As an example, a $C1 stored in $0400 puts an uppercase "A" in the upper left-hand screen position. But, the screen locations aren't mapped in an obvious order, and you get into real hassles over carriage returns and scrolls. There is also no simple way to get a hard copy of a direct screen store. So, direct storing to the screen is usually limited to game scores, status lines, and special effects, rather than being a mainstream way of doing things.

Since outputting text is so important, there are two subroutines built into the Apple's monitor, designed to do most text outputting tasks in the way that most people want them done. One subroutine is called COUT and is located at $FDED. This subroutine will output characters to anything that is connected to the Apple by way of two *character hooks* called CSWL and CSWH and located at $0036 and $0037.

Normally, DOS grabs these character hooks so that it can intercept all output commands, just in case there is something intended for the disk. In turn, DOS will take whatever was plugged into the output hooks, and then plug these into itself.

For instance, a normal hard copy character will get routed from your code to COUT, where it gets passed on to DOS, which checks it for disk commands. The character is then passed on to a printer card, whose code often begins at location $C100. The code will then send the proper commands to the printer itself to print the character. Finally, if you want it to, the printer card code will echo the character on to the screen subroutine.

Which is very slow and roundabout. But this is the standard way of outputting characters that can be routed to DOS, a printer, the screen, or anywhere else you like. This process is extremely slow on the IIe when 80-column firmware is in use. So slow in fact, that you cannot keep up with a 1200-baud modem and scroll the screen at the same time.

The actual *screen subroutine* that puts the characters on the screen is called COUT1 and sits at $FDF0. Good old "Fideyfoo." Fideyfoo automatically keeps track of the horizontal and vertical character positions, does scrolls, handles carriage returns, inverses, your choice of flashing or lowercase, and takes care of most screen actions in the way that most people want most of the time.

Fideyfoo has some locations on page zero reserved that let you pick up special effects quickly and simply. For instance, the size of the scrolling window is set by locations $20 through $24. The cursor hori-

zontal and vertical position bytes CH and CV are located at $24 and $25. Your choice of normal/inverse/flash is decided by INVFLG at $31. And the screen prompt is stashed in $33. See the EMPTY SHELL.SOURCE hooks for other locations of interest.

Here's how to remember when to use COUT or COUT1 . . .

> **Use COUT at $FDED to slowly output a character to DOS, a printer, the screen, or anywhere else you want to send that character. Hooks CSWL and CSWH at $36 and $37 decide where the character is to go.**
>
> **Use COUT1 at $FDF0 to rapidly output a character only to the screen.**

By the way, all these fancy subroutines do take time. It can take half a millisecond just to get through COUT and the DOS code, and any screen scrolls can hold up the works for four or more milliseconds. These times are on older Apples; the IIe is much worse in its 80-column mode. So, it pays to go directly to the screen or output device if speed is important.

It also pays to defeat any "screen echo" should you need top output speed. For instance, a HIRES graphics hard copy dump will be dramatically slowed down if it has to wait for screen scrolling on echo. For fastest possible speed, DOS could also be disconnected during character output times. And fast modems are best used on a IIe in its 40-column or "no-display" modes.

A third place to put your characters involves using a *HIRES character generator* to put your characters onto the HIRES screen. This type of subroutine lets you mix and match graphics and lets you use lots of different text fonts of varying sizes. You can also use special characters to do animation with a HIRES character generator, since your letter "G" is free to look like a frog's face, rather than a stock character. But HIRES character generators are usually rather slow and take bunches of extra code inside your machine.

Normally, a HIRES character generator will grab the COUT hooks "behind" DOS. Its use, once installed, will be pretty much the same as using COUT. Naturally, a HIRES character generator only will display on a HIRES screen and COUT1 will only display on a text screen.

A final place to put characters is to route them to your own custom code subroutine. This lets you rearrange things to suit yourself. A word processor is one example, where the messages all change from use to use. A second example could be a special effects screen filter. This one could "print" in oddball directions, and include delay, sound effects, replacements, screen locking, whole-word breaks, column justify, and most anything else you'd care to dream up.

Normally, you should avoid writing your own code if it more or less duplicates what is already available as ready-to-go subroutines in the Apple monitor. But special code can do special things special ways, and sometimes can give you a tremendous programming advantage over competitive programs.

An edge, even.

So, your first problem is to decide where your text is going to go.

Then, you have to pick some method of getting text to that destination.

Here are the names of several more popular text outputting methods, going from simple to complex . . .

TEXT OUTPUTTING SCHEMES
Brute Force Short File Long File Imbedded Text Compacted

The *brute force* method is simple and obvious. "Give me a D!" "Give me an O!" "Give me a G!" And whaddaya got? A doggedly cumbersome and very painful way to output text. Load the accumulator with the ASCII character for a D and then JSR your output code. Then load the accumulator with an ASCII "O," and so on.

This method is so painful, that you would only want to use it for a four-letter or shorter message, and then if that message was the only one in the program. Among other problems, note that five bytes of code are needed per character output.

The *short file* method is almost as obvious as the brute force method. Put your characters in a file. End each message with some marker, say an ASCII $00 or NUL. If you have to, create a second *pointer* file to tell you where each message starts. The short file is usually limited to 256 or fewer total characters.

The short file method uses indexed addressing to pick sequential characters out of a file. For a detailed example, see the text outenblatter in Volume II of *Don Lancaster's Micro Cookbook* (Sams 21829). Just for kicks, we will also use the short file method in the card shuffler of Ripoff Module 8.

The short file method is limited to a few very short and fixed text messages. But it is quick and simple to program, and may be all you need.

The *long file* method removes the 256 character restriction, by replacing 8-bit indexed loads with 16-bit indirect indexed loads. Your messages can now be any length and you can have any number of them, although there is a slight complication for more than 128 *different* messages at any time.

The long file method is more or less the "standard" way of handling medium length text messages, and is what this ripoff module is all about. We'll find out how your assembler can automate keeping track of messages and message pointers, as well as automatically entering ASCII characters for you.

But, there are limits to the long file method. All your messages must be known and all must be placed at one point in your code. The *imbedded text* method of the next ripoff module very elegantly gets around these restrictions, by letting you put any message you want, any place you want, *directly in your source code.* You can mix and match messages from any modules in your program, so long as one "un-imbedding" subroutine is provided somewhere in your code.

Both the long file and the imbedded text methods take around a byte per character for longer messages. You can remove the end marker on each string message if you switch from high ASCII to low ASCII on the last character. EDASM can do this for you as a special feature. But this complication doesn't save you very much, particularly on longer messages. You can also use a character count byte if you like. Again, this doesn't help much.

Should you have to really cram long messages into your Apple, you can either use repeated disk access or else use some *text compaction* scheme. Repeated disk access is very poor form these days and should be avoided, even with the newer DOS speedup tricks. Text compaction works by using some non-ASCII code that is more efficient than ASCII for character storage.

For instance, in the *Zork* adventures, three characters are crammed into two bytes, giving you code that needs only 67 percent of the space needed by ASCII. In the Adam's version of *Collossial Cave*, letters are arranged into pairs and then each pair is given an unique code. This results in nearly a 50 percent compaction. In spelling checkers, special codes are used to tell how many characters have not changed from the previous character. Special codes are also used for stock endings.

In general, you should not use text compaction until after you are sure you absolutely must have it. It's usually best to have your code completely debugged and your messages completely fixed before using compaction. Note that text compaction will actually *lengthen* and complicate the code needed for short messages, so there is some minimum "breakeven" code length before compaction gains you anything at all.

To recap, there are many places you can put characters and many different ways to generate text messages. One standard way is the text file method, which we will look at here. After that, in Ripoff Module 2, we will check into a more elegant imbedded text method that often is a better choice. Either of these methods is a good choice for your typical "medium" text message jobs, those not so trivial and short that you can handle with obvious code, nor those messages so long that you have to compact them.

So, without further ado, here is . . .
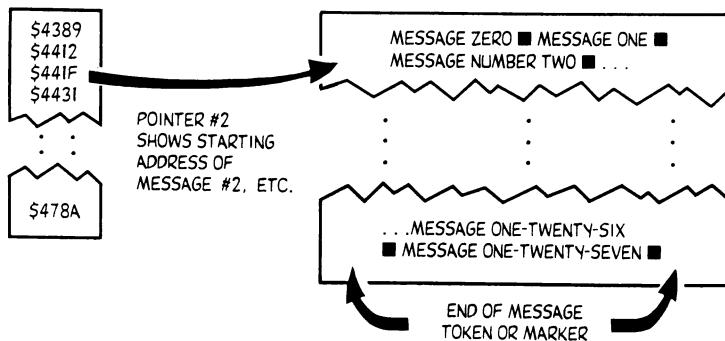
THE LONG FILE METHOD

There are two files involved in the long file method. One of these is called the *pointer file* and the other is called the *message file . . .*

USING A PAIR OF FILES TO OUTPUT TEXT STRINGS:

THE **POINTER FILE** HOLDS THE
16-BIT STARTING ADDRESS OF EACH
TEXT MESSAGE IN THE MESSAGE FILE. . .

THE **MESSAGE FILE** HOLDS ALL OF
THE ASCII TEXT MESSAGES IN SOME
KNOWN ORDER. . .

$4389
$4412
$441F
$4431

POINTER #2
SHOWS STARTING
ADDRESS OF
MESSAGE #2, ETC.

$478A

MESSAGE ZERO ■ MESSAGE ONE ■
MESSAGE NUMBER TWO ■ . . .

. . .MESSAGE ONE-TWENTY-SIX
■ MESSAGE ONE-TWENTY-SEVEN ■

END OF MESSAGE
TOKEN OR MARKER

. . .MESSAGES CAN BE ACCESSED IN ANY ORDER. MORE THAN ONE
POINTER CAN POINT TO THE SAME MESSAGE. EACH MESSAGE
CAN BE ANY LENGTH.

The long file method seems complicated at first, but this text output-ting scheme lets you have messages of any length, and the messages can easily cross 256-byte page boundaries. You can also use different sets of pointer files and text files with the same FLPRINT subroutine.

The message file holds all the messages. The messages do not have to be in any particular order, but the order must be known. Each message ends with a marker of some sort. We will use an ASCII double zero NUL command, since it is easier to test for zero than for any other value. Normally, each message will follow the previous one, although this is not essential. Should you want to put a DOS message into your message file, you start the DOS message with a carriage return and a [D], or "<CTRL> D," otherwise known as an ASCII CR and EOT.

The pointer file holds a list of addresses that show the start of each message. Note that each pointer has to be a 16-bit, or two-byte, address, since the message file can be many pages long. Each pointer file is thus limited to 128 different message pointers, but you can have as many pointer files in your program as you like.

As you might guess, it can be a real drag building and connecting your files by hand. We will show you a fully automatic way to let your assembler build and link files for you. It's all done with creative use of labels.

To use the long file method, you first pick a pointer file. Then you decide which message you want. Say it is message 2. Then you read the pointer file to find the start of the message, say $441F. Then you reach into the message file, starting at $441F, get a character, and then output that character. You continue the process one character at a time till the marker comes up. Then you quit.

The messages do not have to be in any special order, and you can let several different pointers lead you to the same message. This gets handy for prompts like "Please make another selection;" or "That's not a letter, turkey!" defaults or error traps. You also can start at the *middle* of another message, and read to the end. This trick can some-times save you space by using words over again.
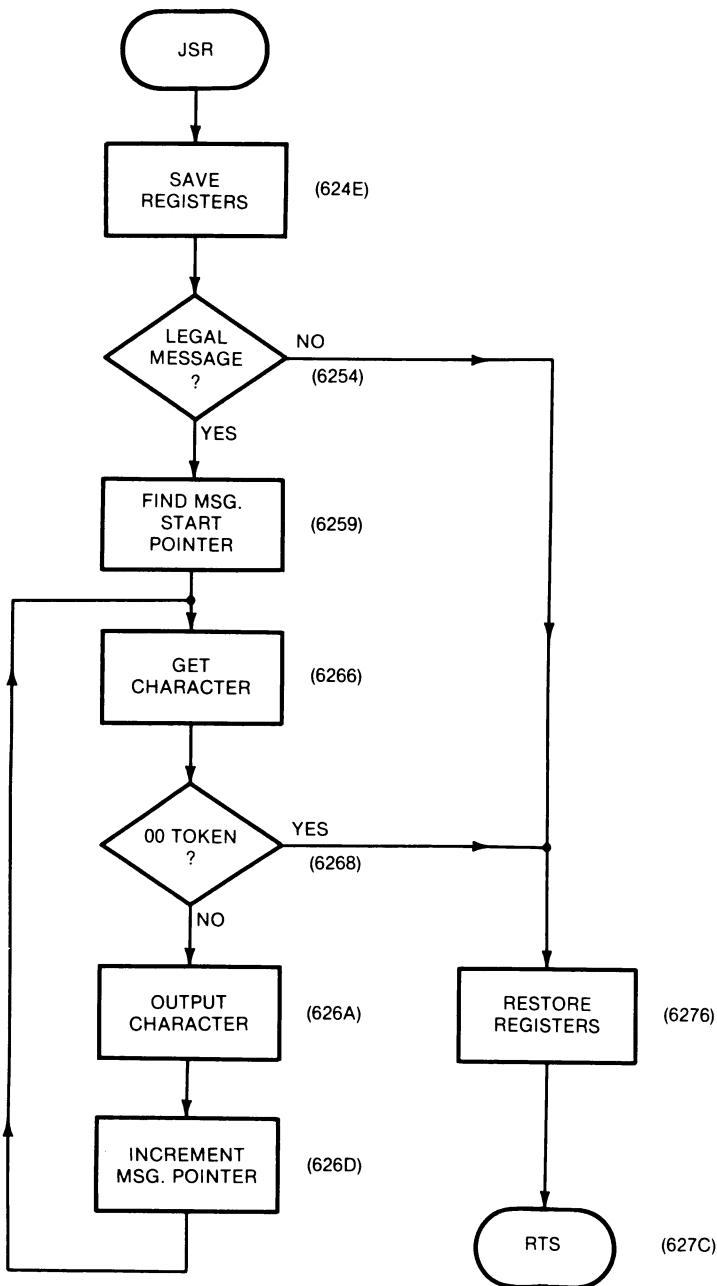
The tricky part is being able to read long messages that cross page boundaries. To do this, you use the powerful 6502 indirect indexed command. In this ripoff module, we will set aside a pair of page zero address locations at $EB and $EC. When we decide to output a message, you reach into the pointer file and put the low half of the message starting address into $EB, and the high half of the message start into $EC.

Then, you set your Y register to #00, and use the LDA($EB),Y indexed indirect addressing instruction. What this command does is go to the sum of the 16-bit address in $EB and $EC (the start of your message) *plus* the Y register value (zero) to get the character to be output.

After the first character, you have a choice. You could increment Y to get to the next character, or else you could add one to the $EB,EC pair. While adding one to Y seems faster and more attractive at first, this will only let you have 256 characters in any one message. So, we will keep Y at zero, and increment the base address. To increment a base address, you first increment the low byte at $EB. If you get a zero result, you then also increment the high byte at $EC. This way, you can continually work your way through most of the 64K address space, without any worries about page boundaries or running out of 8-bit range.

Holding the Y register at zero during an indexed indirect load simply "downgrades" the load command into a straight indirect load. Incidentally, the new 65C02s have "pure" or "unindexed" indirect commands that free up the Y register for other uses.

Confused? Here's a flowchart . . .

FLPRINT FLOWCHART:

Let's check into the actual code of the FLPRINT module, sitting at $624B. This is the module that outputs the text messages for you, and is what you will want to adapt to your own needs.

One good starting place to analyze any code is to find out where variables are stashed. On FLPRINT, we set aside two page zero locations at $EB and $EC to point to the start of our pointer file. These we call PFP1 and PFP1+1. We set aside two more locations at $ED and $EE to use as a running character pointer that works through the message file. These two are labeled MSP1 and MSP1+1.

We also provide a short stash at the end of the subroutine. Three locations here are used for a temporary Y register save YSAV1, an X

register save XSAV1 and a total number-of-messages value at MNUM1.

You enter this FLPRINT module with the message number in the accumulator. You also must have pre-placed the pointer file starting address in PFP1.

We first save the X and Y registers into temporary stashes at XSAV1 and YSAV1. Next, you run a range check of the message number against the stash at MNUM1. A range check makes sure the message is a legal one. This keeps you from outputting garbage or plowing up a disk. We have used a MNUM1 value of $10, good for 16 separate messages.

If the message number is illegal, you restore the X and Y registers and exit without doing anything else. In a "real" program, you would error trap this and do something about it instead.

If the message number is valid, you double it with an ASL, since you are after *pairs* of addresses in the pointer file, each of which takes up 2 bytes.

Then, you reach into the pointer file and get the low half of the address and stash it at MSP1 and then grab the high half and dump it into MSP1+1. We knew *which* pointer file to go to, since whatever code that JSRed here put the pointer file starting address into PFP1 ahead of time.

At this point in the subroutine, we have placed an address into MSP1 and MSP1+1 that points to the first character in the desired message. Now, it's up to the service loop called NXTCHR1 to handle characters for us. NXTCHR1 first grabs a character. If that character was a double zero, the loop quits and exits via END1. This is how you end a message.

Usually, though, the character that NXTCHR1 grabs is not a double zero, so NXTCHR1 passes the character out to the Apple monitor sub-routine at COUT that sends the character to whatever is connected to the output hooks.

Typically, the "hooked " character may go through DOS, which checks it for a [return] [D] header. If it doesn't look like something DOS is interested in, DOS then passes the character somewhere else, possibly to a printer card whose code may start at $C100. The printer card will send the character to a printer, and, optionally, will pass it on to the screen subroutine $FDF0 at COUT1.

None of which matters to NXTCHR1, for once this loop outputs a character to COUT, it couldn't care less what happens to the charac-ter. After the character is sent to wherever it is supposed to go, COUT returns control back to NXTCHR1 via a RTS subroutine return.

NXTCHR1's next job is to move the message file character pointer MSP1 over to the next character. Since this pointer is 16 bits wide, the low byte at MSP1 is first incremented. Should we get a zero result, indicating that a carry is needed to the high byte, we then increment the high byte. This is a pretty much standard way of incrementing a 16-bit address pointer pair.

Following that, we jump back to the start of the NXTCHR1 loop and keep outputting characters till we hit the double zero.

Note the forced branch at NOC1. It pays to keep absolute jumps out of any of your code modules, for absolute references make code harder to relocate. The CLC and BCC commands together do an unconditional relative branch for you that is easily relocatable.

After the double zero, we get out of the FLPRINT subroutine by restoring the X and Y registers and doing the usual RTS back to whoever it was that JSRed this module.

The DEMO1 that starts at $6200 is a rather unexciting "exerciser" that shows us how FLPRINT works. DEMO1 first sets the total number of messages to $03 and then finds out where the message pointer file sits. It then stores the pointer file start in MSP1, for use by FLPRINT.

Then we clear the screen, do some tabbing, and go to the inverse mode. Message #00 is called for, and gotten through FLPRINT. We return to normal text for message #01. Note that message #02 is quite long. It could, in fact, be any length you want, within the limits of available memory.

After message #01, we ask for user input. Should we get an "E," we exit the program. A "C" gives you a DOS catalog. This is done by printing first a CR and then an EOT, or [D], followed by the CATALOG string. If the CATALOG is long enough, extra prompts are needed for each catalog page.

Should you enter anything but an "E" or a "C," the entire FLPRINT module is rerun. This error trapping causes a brief flash on the screen, which should be enough of an operator "hey turkey!" prompt for most users.

In this example, we require a capital C or capital E. It is better practice to allow for either uppercase or lowercase entries. You can do this with a double test, or else by forcing lowercase characters into their uppercase equivalents. This important detail should not be omitted on the IIe or for older uses where you expect mixed cases. You'll find a case changer example in Ripoff Module 7.

Note that both the pointer file and the message file can be repeatedly reloaded off the disk. Thus, there is no limit to using external or calculated text strings as might be needed in a longer adventure.

Creating the Files

A good assembler will very much simplify setting up and creating your own pointer and text files. The process of putting the file into memory and properly linking it with everything else can be made fully automatic, without any worries about absolute addresses.

It's labels to the rescue.

Let's look at the message file first. First and foremost, you put a label at the beginning of each separate message. We have used M1.0 through M1.15 in the source code. If you have a label on your message, your assembler can find the message, regardless of where it ends up in memory.

We stopped at sixteen messages only to save on source code length. You can make things as long as you like, with up to 128 messages for each message pointer file and as many message pointer files as you want.

The ASC and DFB commands greatly simplify entering your messages. We've done almost everything here in uppercase for compatibility with older Apples, but most newer assemblers will let you use full case for your messages. "New way" editing also lets you do lowercase on most any assembler.

The ASC command tells the assembler to "convert what follows to ASCII." High ASCII is normally used in the Apple, although you can

change this if you want to. While each ASCII string can be any length, it pays to keep each string under 32 characters or so. This makes for neater assembly listings. You can tie as many strings together as you need to get the total message.

A delimiter should start the ASCII text string. Use a quote for the delimiter unless you really want to print a quote. Then use a slash instead. An ending delimiter is not needed if there are no comments on the string line. Usually there won't be room for comments anyway, so this is no big deal. This is roughly similar to not needing the final quote in an Applesloth PRINT statement. Trailing spaces are hard to see without a final delimiter.

So much for alphanumerics. How do you handle control characters?

Obviously, you need a way to, say, imbed carriage returns. Yet if you type a carriage return, the string command completes itself. How do you get out of this bind?

Once again, it's labels to the rescue. Just as you can use a CHR$(13) to fool a higher level language into outputting a carriage return, you can trick an assembler into entering a carriage return into a file by using a label.

I've chosen to use single letter labels for control commands. B for backspace, C for carriage return, D for DOS, and X for the NULL or double zero. Each letter must be pre-defined as a constant, such as a B EQU $88 for a backspace. To enter control commands into your ASCII text, simply use DFBs with as many control commands as you like.

For instance, a DFB C,X puts a carriage return and an end-of-text marker into your message file. That wild DFB B,B,B,P,B,X sequence uses backspaces to center a flashing user prompt inside a fancy screen symbol. Incidentally, this may look different on a II and IIe. You might like to change it per the Apple in use.

Unfortunately, this was written before "new" EDASM became available. Since "A," "X," and "Y" are disallowed labels in "new" EDASM, you'll have to substitute something else for "X." Note that the STR pseudo-op in "new" EDASM can eliminate any need for a trailing NULL.

Summing up our message file, be sure to put a label on the start of each message. Then enter your ASCII characters using the assembler's ASC command. Don't forget the delimiter at the front and don't let the individual ASC strings get too long. Enter any control characters you want to imbed with DFB commands. On a typical message, you will alternate ASC and DFB commands. Use ASC for the letters and DFB for the carriage returns and end markers.

The pointer file will usually be much shorter than the message file. The pointer file holds the starting address of each message, so that FLPRINT knows where to go to start outputting characters.

To automate the construction of a pointer file, just use labels for each pointer entry. For instance, we call the pointer to the sixth message PF5 (don't forget that zero!). Our pointer source code under label PF5 tells us to DW M1.5, or to "go to wherever the message labeled M1.5 happens to be, find its present absolute address, and put that address pair back here."

Which is an awful lot of work for the assembly program. But that's its job and is one of the many reasons why we use an assembler in the first place—to automate most of the dogwork involved in writing machine language programs.

MIND BENDERS

—Show how FLPRINT can be simplified if you only have one pointer file in your program.

—FLPRINT works fine when called from *within* another program, but there's a slight bug when used directly from the monitor or Applesoft. What is the bug? What causes it? How can you prevent it?

—How can you design a program that outputs lowercase only to those machines that can use it?

—Can FLPRINT be used with changing messages? How?

—Show ways to use FLPRINT with several different pointer files.

—Rewrite this module to use "new" EDASM's string command STR, which includes a message count byte. What are the advantages of this new method?

PROGRAM RM-1
FILE BASED PRINTER

```
----- NEXT OBJECT FILE NAME IS FLPRINT
6200:            3           ORG  $6200      ; PUT MODULE #1 AT $6200


6200:            5 ;   *******************************************
6200:            6 ;   *                                         *
6200:            7 ;   *            -< FLPRINT MODULE >-          *
6200:            8 ;   *                                         *
6200:            9 ;   *       (FILE BASED STRING PRINTER)       *
6200:           10 ;   *                                         *
6200:           11 ;   *       VERSION 1.0   ($6200-$642A)       *
6200:           12 ;   *                                         *
6200:           13 ;   *                6-15-83                  *
6200:           14 ;   *.......................................* 
6200:           15 ;   *                                         *
6200:           16 ;   *          COPYRIGHT C 1983 BY            *
6200:           17 ;   *                                         *
6200:           18 ;   *      DON LANCASTER AND SYNERGETICS      *
6200:           19 ;   *      BOX 1300, THATCHER AZ., 85552      *
6200:           20 ;   *                                         *
6200:           21 ;   *      ALL COMMERCIAL RIGHTS RESERVED     *
6200:           22 ;   *                                         *
6200:           23 ;   *******************************************


6200:           25 ;            *** WHAT IT DOES ***

6200:           27 ;   THIS MODULE OUTPUTS TEXT STRINGS OR DOS COMMANDS
6200:           28 ;   TO THE APPLE II'S OUTPUT HOOKS, USING STRINGS
6200:           29 ;   THAT ARE COLLECTED TOGETHER IN A COMMON FILE.
6200:           30 ;
6200:           31 ;
6200:           32 ;


6200:           34 ;            *** HOW TO USE IT ***

6200:           36 ;   YOUR CALLING CODE SHOULD HAVE PREVIOUSLY STORED
6200:           37 ;   A MESSAGE POINTER FILE ADDRESS IN PFP1 (LOW) AND
6200:           38 ;   PFP1+1 (HIGH).  ONE OF 128 POSSIBLE RESPONSES
6200:           39 ;   ARE SELECTED BY LOADING THE ACCUMULATOR WITH A
6200:           40 ;   MESSAGE NUMBER AND THEN DOING A JSR TO FLPRINT.
6200:           41 ;
```

PROGRAM RM-1, CONT'D . . .

```
6200:              44 ;              *** GOTCHAS ***

6200:              46 ;      THIS METHOD IS BEST USED FOR LONG MESSAGES THAT MIGHT
6200:              47 ;      NEED CALCULATED VALUES OR DISK-BASED CHANGES.
6200:              48 ;
6200:              49 ;      MESSAGES CAN BE ANY LENGTH, BUT MORE THAN 128 DIFFERENT
6200:              50 ;      MESSAGES WILL NEED SEPARATE MSP1 ADDRESS BASES.   EACH
6200:              51 ;      MESSAGE MUST END IN A $00 MARKER.


6200:              53 ;              *** ENHANCEMENTS ***

6200:              55 ;      DOS COMMANDS ARE OUTPUT BY STARTING THE STRING
6200:              56 ;      WITH A CARRIAGE RETURN AND <CTRL> D.
6200:              57 ;
6200:              58 ;      TO GO DIRECTLY TO THE SCREEN, USE COUT1 RATHER THAN COUT.
6200:              59 ;      THIS IS FASTER, BUT CANNOT CONTROL DOS OR BE PRINTED.
6200:              60 ;


6200:              62 ;              *** RANDOM COMMENTS ***

6200:              64 ;      TO RUN THE DEMO, USE $6200G OR CALL 25088.
6200:              65 ;
6200:              66 ;      THE X AND Y REGISTERS ARE PRESERVED; A IS DESTROYED.
6200:              67 ;
6200:              68 ;
6200:              69 ;
```

PROGRAM RM-1, CONT'D . . .

```
6200:          72 ;              *** HOOKS ***


FDED:          74 COUT     EQU  $FDED    ; OUTPUT CHARACTER VIA HOOKS
FC58:          75 HOME     EQU  $FC58    ; CLEAR SCREEN
FB2F:          76 INIT     EQU  $FB2F    ; INITIALIZE TEXT SCREEN
C010:          77 KBDSTR   EQU  $C010    ; KEYBOARD RESET
F94A:          78 PRBL2    EQU  $F94A    ; PRINT X BLANKS
FD0C:          79 RDKEY    EQU  $FD0C    ; GET INPUT CHARACTER
FE80:          80 SETINV   EQU  $FE80    ; SET INVERSE SCREEN
FE84:          81 SETNORM  EQU  $FE84    ; SET NORMAL SCREEN


00ED:          83 MSP1     EQU  $ED      ; MESSAGE FILE CHARACTER POINTER
00EB:          84 PFP1     EQU  $EB      ; POINTER FILE STARTING ADDRESS




6200:          86 ;              *** TEXTFILE COMMANDS ***


0088:          88 B        EQU  $88      ; BACKSPACE
008D:          89 C        EQU  $8D      ; CARRIAGE RETURN
0084:          90 D        EQU  $84      ; DOS ATTENTION
0060:          91 P        EQU  $60      ; FLASHING PROMPT
0000:          92 X        EQU  $00      ; END OF MESSAGE
```

PROGRAM RM-1, CONT'D . . .

```
6200:          95 ;               *** DEMO ***
6200:          96 ;


6200:          98 ;        THE DEMO USES THE FLPRINT MODULE TO OUTPUT
6200:          99 ;        SCREEN MESSAGES AND A DOS CATALOG COMMAND.
6200:         100 ;


6200:A9 03    102 DEMO1  LDA  #$03     ; THREE MESSAGES TOTAL
6202:8D 7D 62 103        STA  MNUM1    ; SAVE FOR CHECK
6205:A9 80    104        LDA  #>PF0    ; SAVE MESSAGE POINTER LOW
6207:85 EB    105        STA  PFP1     ;
6209:A9 62    106        LDA  #<PF0    ; SAVE MESSAGE POINTER HIGH
620B:85 EC    107        STA  PFP1+1   ;

620D:20 2F FB 109        JSR  INIT     ; GO TO TEXT MODE
6210:20 58 FC 110        JSR  HOME     ; CLEAR SCREEN
6213:A2 08    111        LDX  #$08     ; PRINT BLANKS VIA MONITOR
6215:20 4A F9 112        JSR  PRBL2    ;

6218:20 80 FE 114        JSR  SETINV   ; INVERSE TEXT FOR TITLE
621B:A9 00    115        LDA  #00      ; MESSAGE #0
621D:20 4E 62 116        JSR  FLPRINT  ; PRINT MESSAGE

6220:20 84 FE 118        JSR  SETNORM  ; NORMAL TEXT
6223:A9 01    119        LDA  #$01     ; MESSAGE #1
6225:20 4E 62 120        JSR  FLPRINT  ; PRINT MESSAGE

6228:2C 10 C0 122        BIT  KBDSTR   ; RESET KEYBOARD
622B:20 0C FD 123        JSR  RDKEY    ; GET KEY
622E:C9 C5    124        CMP  #$C5     ; AN "E" FOR EXIT?
6230:F0 15    125        BEQ  EXIT1    ;   YES, EXIT
6232:C9 C3    126        CMP  #$C3     ; A "C" FOR CATALOG?
6234:D0 CA    127        BNE  DEMO1    ; TRY AGAIN FOR VALID KEY

6236:20 58 FC 129        JSR  HOME     ; CLEAR SCREEN, THEN
6239:A9 02    130        LDA  #$02     ; DO CATALOG
623B:20 4E 62 131        JSR  FLPRINT  ;
623E:2C 10 C0 132        BIT  KBDSTR   ; HOLD CATALOG
6241:20 0C FD 133        JSR  RDKEY    ;   TILL KEYPRESS

6244:18       135        CLC           ; BRANCH ALWAYS
6245:90 B9    136        BCC  DEMO1    ;   AND TRY AGAIN

6247:20 58 FC 138 EXIT1  JSR  HOME     ; EXIT DEMO
624A:2C 10 C0 139        BIT  KBDSTR   ; RESET KEYBOARD
624D:60       140        RTS           ;
```

PROGRAM RM-1, CONT'D . . .

```
624E:          143 ;                *** FLPRINT MODULE ***
624E:          144 ;
624E:          145 ;


624E:          147 ;            THIS MODULE USES THE ACCUMULATOR VALUE TO
624E:          148 ;            FIND A POINTER TO THE TEXT STRING. IT THEN
624E:          149 ;            OUTPUTS ONE CHARACTER AT A TIME TILL THE $00
624E:          150 ;            END-OF-MESSAGE MARKER IS FOUND.
624F:          151 ;
624E:          152 ;


624E:8C 7F 62  154 FLPRINT STY YSAV1     ; SAVE REGISTERS
6251:8E 7E 62  155         STX XSAV1     ;

6254:CD 7D 62  157         CMP MNUM1     ; A LEGAL MESSAGE NUMBER?
6257:B0 1D     158         BCS END1      ; DON'T PRINT IF ILLEGAL
6259:0A        159         ASL A         ; DOUBLE POINTER FOR ADDRESS PAIR
625A:A8        160         TAY           ;
625B:B1 EB     161         LDA (PFP1),Y  ; GET LOW POINTER
625D:85 ED     162         STA MSP1      ;   AND SAVE
625F:C8        163         INY           ;
6260:B1 EB     164         LDA (PFP1),Y  ; GET HIGH POINTER
6262:85 EE     165         STA MSP1+1    ;   AND SAVE
6264:A0 00     166         LDY #$00      ; NO INDEXING
6266:B1 ED     167 NXTCHR1 LDA (MSP1),Y  ; GET CHARACTER
6268:F0 0C     168         BEQ END1      ; EXIT ON $00 MARKER
626A:20 ED FD  169         JSR COUT      ; PRINT CHARACTER

626D:E6 ED     171         INC MSP1      ; CALCULATE NEXT CHARACTER LOCATION
626F:D0 02     172         BNE NOC1      ;   IF A CARRY, THEN
6271:E6 EE     173         INC MSP1+1    ; INCREMENT HIGH ADDRESS LOCATION
6273:18        174 NOC1    CLC           ; BRANCH ALWAYS TO
6274:90 F0     175         BCC NXTCHR1   ;   GET NEXT CHARACTER

6276:AE 7E 62  177 END1    LDX XSAV1     ; RESTORE REGISTERS
6279:AC 7F 62  178         LDY YSAV1     ;
627C:60        179         RTS           ; AND EXIT




627D:          181 ;                *** STASH ***



627D:10        184 MNUM1   DFB $10       ; NUMBER OF MESSAGES IN FILE
627E:00        185 XSAV1   DFD $00       ; X-REGISTER SAVE
627F:00        186 YSAV1   DFB $00       ; Y-REGISTER SAVE
```

PROGRAM RM-1, CONT'D . . .

```
6280:                189 ;                 *** POINTER FILE ***


6280:A0 62           191 PF0      DW       M1.0       ; POINTER FILE
6282:B6 62           192 PF1      DW       M1.1       ;
6284:FA 63           193 PF2      DW       M1.2       ;
6286:05 64           194 PF3      DW       M1.3       ;
6288:08 64           195 PF4      DW       M1.4       ;
628A:0B 64           196 PF5      DW       M1.5       ;
628C:0E 64           197 PF6      DW       M1.6       ;
628E:11 64           198 PF7      DW       M1.7       ;
6290:14 64           199 PF8      DW       M1.8       ;
6292:17 64           200 PF9      DW       M1.9       ;
6294:1A 64           201 PF10     DW       M1.10      ;
6296:1D 64           202 PF11     DW       M1.11      ;
6298:20 64           203 PF12     DW       M1.12      ;
629A:23 64           204 PF13     DW       M1.13      ;
629C:26 64           205 PF14     DW       M1.14      ;
629E:29 64           206 PF15     DW       M1.15      ;



62A0:                208 ;                 *** MESSAGE FILE ***


62A0:CD C5 D3        210 M1.0     ASC                "MESSAGE FILE METHOD"
62A3:D3 C1 C7
62A6:C5 A0 C6
62A9:C9 CC C5
62AC:A0 CD C5
62AF:D4 C8 CF
62B2:C4
62B3:8D 8D 00        211          DFB                C,C,X

62B6:D7 C9 D4        213 M1.1     ASC                "WITH THIS METHOD,   ALL OF THE MESSAGES
62B9:C8 A0 D4
62BC:C8 C9 D3
62BF:A0 CD C5
62C2:D4 C8 CF
62C5:C4 AC A0
62C8:A0 C1 CC
62CB:CC A0 CF
62CE:C6 A0 D4
62D1:C8 C5 A0
62D4:CD C5 D3
62D7:D3 C1 C7
62DA:C5 D3 A0
62DD:8D              214          DFB                C
```

PROGRAM RM-1, CONT'D . . .

```
62DE:C1 D2 C5    217         ASC         "ARE COMBINED INTO A COMMON TEXT FILE
62E1:A0 C3 CF
62E4:CD C2 C9
62E7:CE C5 C4
62EA:A0 C9 CE
62ED:D4 CF A0
62F0:C1 A0 C3
62F3:CF CD CD
62F6:CF CE A0
62F9:D4 C5 D8
62FC:D4 A0 C6
62FF:C9 CC C5
6302:AE
6303:8D 8D       218         DFB         C,C

6305:C1 A0 D0    220         ASC         "A POINTER FILE IS USED TO DECIDE WHICH
6308:CF C9 CE
630B:D4 C5 D2
630E:A0 C6 C9
6311:CC C5 A0
6314:C9 D3 A0
6317:D5 D3 C5
631A:C4 A0 D4
631D:CF A0 C4
6320:C5 C3 C9
6323:C4 C5 A0
6326:D7 C8 C9
6329:C3 C8 A0
632C:8D          221         DFB         C

632D:CD C5 D3    223         ASC         "MESSAGE IS TO BE OUTPUT.
6330:D3 C1 C7
6333:C5 A0 C9
6336:D3 A0 D4
6339:CF A0 C2
633C:C5 A0 CF
633F:D5 D4 D0
6342:D5 D4 AE
6345:8D 8D       224         DFB         C,C

6347:D5 D3 C5    226         ASC         "USES INCLUDE TEXT DATA BASES AND OTHER
634A:D3 A0 C9
634D:CE C3 CC
6350:D5 C4 C5
6353:A0 D4 C5
6356:D8 D4 A0
6359:C4 C1 D4
635C:C1 A0 C2
635F:C1 D3 C5
6362:D3 A0 C1
6365:CE C4 A0
6368:CF D4 C8
636B:C5 D2
636D:8D          227         DFB         C
```

PROGRAM RM-1, CONT'D . . .

```
636E:D0 CC C1   230        ASC        "PLACES WHERE LOTS OF CHANGING MESSAGE
6371:C3 C5 D3
6374:A0 D7 C8
6377:C5 D2 C5
637A:A0 CC CF
637D:D4 D3 A0
6380:CF C6 A0
6383:C3 C8 C1
6386:CE C7 C9
6389:CE C7 A0
638C:CD C5 D3
638F:D3 C1 C7
6392:C5 D3
6394:8D          231        DFB        C

6395:C1 D2 C5   233        ASC        "ARE TO BE PRINTED OR DISPLAYED.
6398:A0 D4 CF
639B:A0 C2 C5
639E:A0 D0 D2
63A1:C9 CE D4
63A4:C5 C4 A0
63A7:CF D2 A0
63AA:C4 C9 D3
63AD:D0 CC C1
63B0:D9 C5 C4
63B3:AE
63B4:8D 8D 8D   234        DFB        C,C,C,C
63B7:8D
63B8:D4 D9 D0   235        ASC        /TYPE "C" FOR CATALOG, OR "E" FOR EXIT
63BB:C5 A0 A2
63BE:C3 A2 A0
63C1:C6 CF D2
63C4:A0 C3 C1
63C7:D4 C1 CC
63CA:CF C7 AC
63CD:A0 CF D2
63D0:A0 A2 C5
63D3:A2 A0 C6
63D6:CF D2 A0
63D9:C5 D8 C9
63DC:D4 AE
63DE:8D 8D 8D   236        DFB        C,C,C,C
63E1:8D
63E2:A0 A0 A0   237        ASC        "          -< >-"
63E5:A0 A0 A0
63E8:A0 A0 A0
63EB:A0 A0 A0
63EE:A0 AD BC
63F1:A0 BE AD
63F4:88 88 88   238        DFB        B,B,B,P,B,X
63F7:60 88 00
```

PROGRAM RM-1, CONT'D . . .

```
63FA:8D 84      241 M1.2    DFB         C,D
63FC:C3 C1 D4   242         ASC         "CATALOG"
63FF:C1 CC CF
6402:C7
6403:8D 00      243         DFB         C,X

6405:A0         245 M1.3    ASC         " "
6406:8D 00      246         DFB         C,X

6408:A0         248 M1.4    ASC         " "
6409:8D 00      249         DFB         C,X

640B:A0         251 M1.5    ASC         " "
640C:8D 00      252         DFB         C,X

640E:A0         254 M1.6    ASC         " "
640F:8D 00      255         DFB         C,X

6411:A0         257 M1.7    ASC         " "
6412:8D 00      258         DFB         C,X

6414:A0         260 M1.8    ASC         " "
6415:8D 00      261         DFB         C,X

6417:A0         263 M1.9    ASC         " "
6418:8D 00      264         DFB         C,X

641A:A0         266 M1.10   ASC         " "
641B:8D 00      267         DFB         C,X

641D:A0         269 M1.11   ASC         " "
641E:8D 00      270         DFB         C,X

6420:A0         272 M1.12   ASC         " "
6421:8D 00      273         DFB         C,X

6423:A0         275 M1.13   ASC         " "
6424:8D 00      276         DFB         C,X

6426:A0         278 M1.14   ASC         " "
6427:8D 00      279         DFB         C,X

6429:A0         281 M1.15   ASC         " "
642A:8D 00      282         DFB         C,X
```

*** SUCCESSFUL ASSEMBLY: NO ERRORS

IMBEDDED STRING PRINTER

**a powerful and very sneaky
way of mixing and matching
text messages**

I guess I've always been attracted to elegant simplicity, particularly when it is combined with sneakiness. The file based text printer of Ripoff Module 1 is a classic and standard old warhorse that's cumbersome, restrictive, and hard to use. It obviously doesn't qualify. What can we do that is better?

Why do we need a text message file at all? Why not, instead, simply *imbed* the text messages *directly into the source code* when and where they are needed? This way, you can have any number of short and fixed messages anywhere in your program, and you can mix and match modules from all over the lot without any worries *at all about* creating a big master text file and bunches of pointers to work with it.
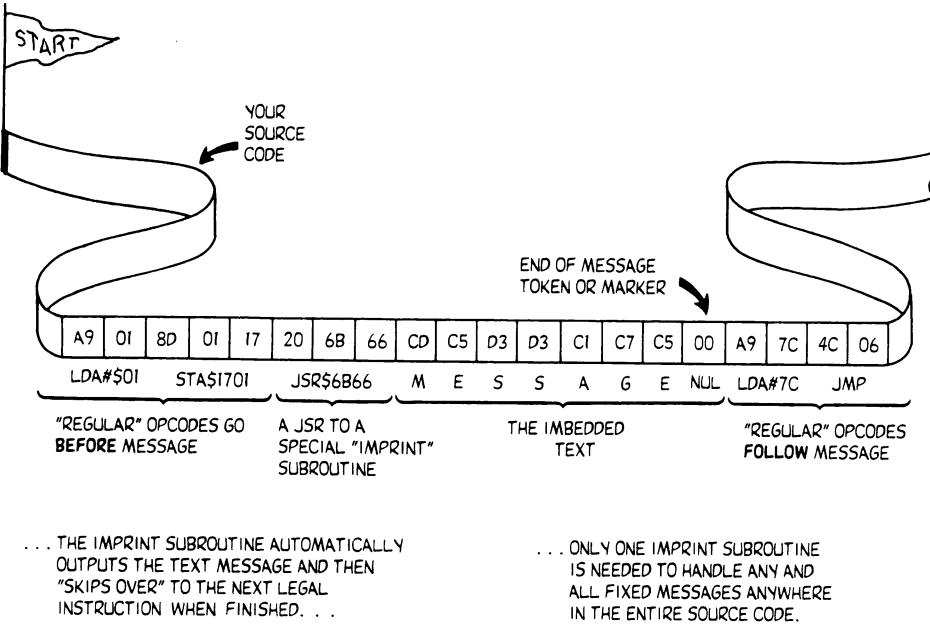
The usual excuse for not imbedding text into source code is that the 6502 tends to get violently ill when you feed it ASCII text instead of machine language commands. The trick is to find some elegantly simple way to keep the imbedded messages out of the CPU. The way is called the *imbedded text* method.

With the imbedded text method, you simply insert ASCII text or DOS strings into your source code when and as you need them. Immediately before the strings, you do a jump to a very special subroutine that will grab all the ASCII stuff for its own use, and then let

the 6502 pick up the machine language commands that *follow* the message.

Like so . . .

HOW TO IMBED TEXT INTO SOURCE CODE:

| A9 | 01 | 8D | 01 | 17 | 20 | 6B | 66 | CD | C5 | D3 | D3 | CI | C7 | C5 | 00 | A9 | 7C | 4C | 06 |

| LDA#$01 | STA$1701 | JSR$6B66 | M E S S A G E NUL | LDA#7C | JMP |

| "REGULAR" OPCODES GO BEFORE MESSAGE | A JSR TO A SPECIAL "IMPRINT" SUBROUTINE | THE IMBEDDED TEXT | "REGULAR" OPCODES FOLLOW MESSAGE |

. . . THE IMPRINT SUBROUTINE AUTOMATICALLY OUTPUTS THE TEXT MESSAGE AND THEN "SKIPS OVER" TO THE NEXT LEGAL INSTRUCTION WHEN FINISHED. . .

. . . ONLY ONE IMPRINT SUBROUTINE IS NEEDED TO HANDLE ANY AND ALL FIXED MESSAGES ANYWHERE IN THE ENTIRE SOURCE CODE.

You will need only one imbedded printing subroutine. This can go anywhere in your program. That sub is called IMPRINT. Any and all program modules can use this lone IMPRINT subroutine any time they want to output a fixed text message. While most of these messages will usually be short, there is essentially no limit, except for memory space, as to how long your messages are, how many messages you use, or how you mix and match them

And all this *without* any pointers or master text files.

IMPRINT works by first finding out who called it. It does this by looking into the stack to find the intended subroutine return address. Not only does IMPRINT find the return address, but it *steals* it off the stack and uses that address as a string pointer. It then increments the return address ASCII character by ASCII character, until the message is finished. Finally, IMPRINT forces a subroutine return that goes *beyond* the imbedded text and picks up on the next mainstream machine lan-*guage command.*
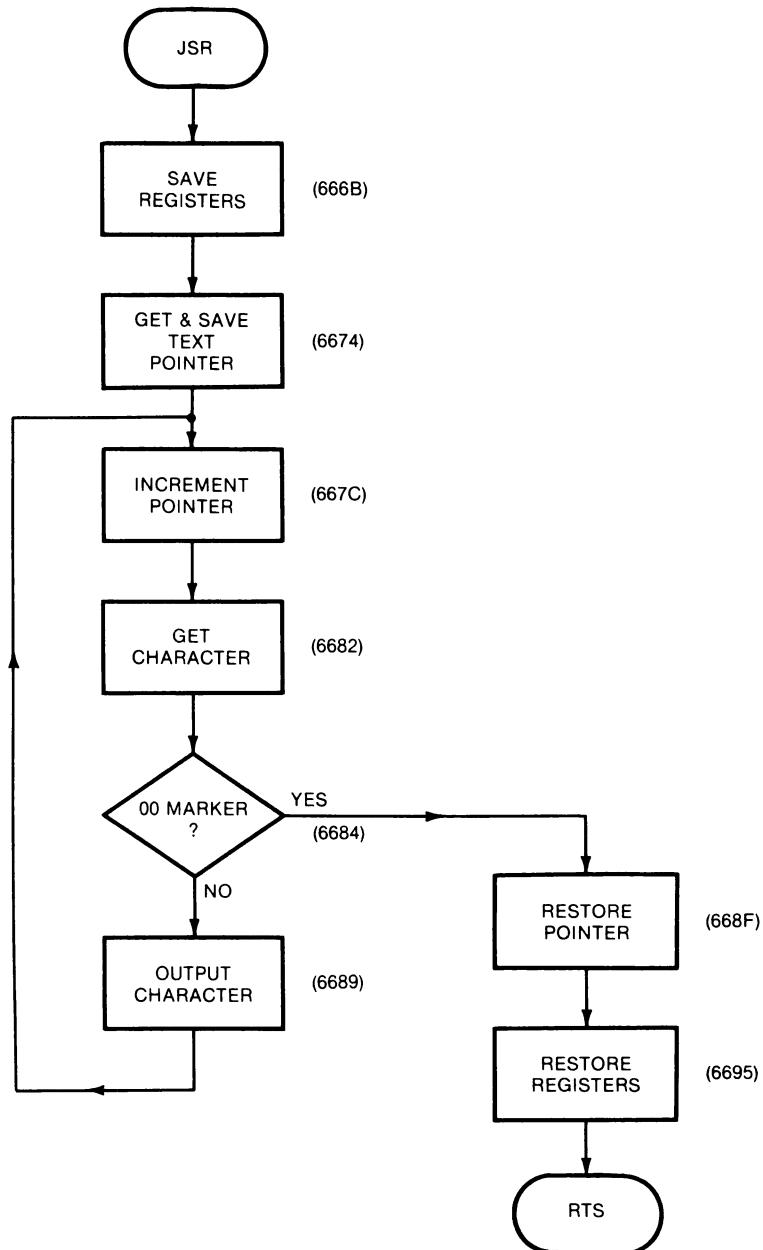
What is elegant and sneaky about the whole thing is that IMPRINT is not really a subroutine at all! IMPRINT is a "mainline" code module that "plugs itself" into high level code when and where it is called. It does this by messing with the stack. First, it pulls the return address off the stack, converting itself into mainline code. When finished output-ting text, IMPRINT pushes the return-to-the-next-machine-language address onto the stack and then does a quick RTS, which is nothing but a forced jump.

Sounds hairy.

And it is. But the code is very short and simple. It's also very easy to use once you understand it. And, as further elegance, IMPRINT does not hurt *any* working registers at all.

Here's a flowchart of IMPRINT . . .

IMPRINT FLOWCHART:

```
                    ┌──────────┐
                    │   JSR    │
                    └────┬─────┘
                         │
                 ┌───────────────┐
                 │    SAVE       │   (666B)
                 │  REGISTERS    │
                 └───────┬───────┘
                         │
                 ┌───────────────┐
                 │  GET & SAVE   │   (6674)
                 │    TEXT       │
                 │   POINTER     │
                 └───────┬───────┘
                         │
                 ┌───────────────┐
                 │  INCREMENT    │   (667C)
                 │   POINTER     │
                 └───────┬───────┘
                         │
                 ┌───────────────┐
                 │    GET        │   (6682)
                 │  CHARACTER    │
                 └───────┬───────┘
                         │
                    ◇ 00 MARKER ◇  YES ──────────┐
                    ◇    ?      ◇   (6684)        │
                         │NO               ┌───────────────┐
                 ┌───────────────┐         │   RESTORE     │   (668F)
                 │   OUTPUT      │ (6689)   │   POINTER     │
                 │  CHARACTER    │         └───────┬───────┘
                 └───────────────┘                 │
                                          ┌───────────────┐
                                          │   RESTORE     │   (6695)
                                          │  REGISTERS    │
                                          └───────┬───────┘
                                              ┌──────────┐
                                              │   RTS    │
                                              └──────────┘
```

IMPRINT sits at $666B right now, but it is easily put any place you want.

As before, to understand a machine language module, find out what variables are stashed where. Two slots on page zero are set aside as a pointer to the character being output. These are called STRP2 and STRP2+1 and are located at $EB and $EC. Three absolute slots are used to save the registers, and are called ASAV2, XSAV2, and YSAV2, and appear as a short stash that follows IMPRINT.

An aside or two. A pair of mnemonics involved in a 16-bit word can be spelled out either as STRP2L and STRP2H, or as STRP2 and STRP2+1. The H and L stand for high and low. The standard way is to

use high and low, but you save on code and EQUs by using the arith-metic addition feature of your assembler. Do an EQU on STRP2, and your STRP2+1 rides along free.

By the way, the 2 tag just stands for module 2. This way, you can combine the ripoff modules anyway you like without worrying about duplicate label errors for common names.

Secondly, there are many different ways to temporarily save your accumulator and X- and Y-registers. It is usually a good idea to save all working registers during a subroutine or service module, so you keep any surprises out of the calling code. We have used absolute stores, since they are the safest and surest way of stashing things without memory conflicts. Absolute stores can take more bytes, can be slower, and are a somewhat harder to relocate than other storage methods. Page zero stores are faster, but you tie up precious and possibly con-flicting real estate when you try this. The stack is another obvious stash, but its use gets messy fast, particularly on code like IMPRINT that purposely messes with the stack.

The absolute *worst* place to save working registers is in the monitor register saving subroutines IOSAVE and IOREST . . .

Don't EVER use the monitor routines IOSAVE and IOREST!

Sooner or later, they are bound to create problems.

What happens is that some module will use IOSAVE for its register saves and then may JSR to some other module that also tries to use IOSAVE for its own use. The first save gets overwritten by the second, and the final IOREST does a self-destruct, rather than a restore.

Let's see. Where were we? Back to IMPRINT. We first save our regis-ters to the three absolute locations ASAV2, XSAV2, and YSAV2, and stash these at the end of the module.

The subroutine return address in the 6502's stack pointer takes two bytes. The *low address* is the first one you get back. The *high address* is the second byte you get back. That address points to *one less* than where you end up . . .

6502 SUBROUTINE STACK RULES

Two bytes on a stack are used to save a subroutine return address.

The FIRST byte you get back holds the return address POSITION byte.

The SECOND byte you get back holds the return address PAGE byte.

The RTS command returns you to the return address PLUS ONE.

So, we grab the top of the stack and store it as the low address half

at STRP2. Then we grab the top of the stack again, and this time store it as the high address half at STRP2+1.

But, note at this time that this "return" address is pointing to *one less than* our first ASCII character, rather than to a "safe" 6502 return point. Note also that *we are no longer in a subroutine*. Why? Because the calling code pushed two things onto the stack, and the using code pulled two things back off of the stack. We are thus once again back in high level code!

To get our string pointer STRP2 pointing to our first ASCII character, we simply increment the pair in the usual way. Do this by incrementing STRP2, and then, if you get a zero result, take care of the overflow by incrementing STRP2+1. Since we know we will have to increment to get between characters, we'll arrange things so we only need one increment command, at the head of the loop called NXTCHH2.

Your ASCII or DOS text string gets entered into your calling source code, and should end with some marker. We will use the ASCII double zero NULL command here, since it is simplest.

At this time, we grab the character from the string using the indirect indexed loading that lets us reach any point in the 16-bit address space without any page boundary worries. As before, we have forced the Y-register to $00, to downgrade the indirect indexed command into a "pure" indirect load.

Having gotten the character, we can test it for a double zero. If we get the double zero, we go on to the exit routine at END2. If not, we output the character to COUT or to Fideyfoo, or wherever.

Next, we have included a JSR to an immediate return that we call HOOK2. This has no present use, but it lets you grab IMPRINT for special effects such as character delay, sound, printing in a weird screen direction, or whatever. To use it, just let the subroutine lead you to your special effects module.

After this unused hook, a relative forced branch that fakes an unconditional jump gets us back to NXTCHR2 and completes the loop.

Processing continues one character at a time until we get to the double zero. Then we branch down to the END2 routine.

At this time, the STRP2 pointer is pointing to the double zero of the last character, which is one less than the address of the continuing machine language code in the mainstream. On a subroutine return, the RTS command always goes to one more than the return address. So, *STRP2 equals the correct subroutine return address* when it is pointing to the end-of-text marker.

All the remains is to get back to the mainstream code. We might be tempted to try using the jump indirect instruction, but this one has a deadly bug that will nail you one time out of 128 . . .

The JMP indirect command has a deadly bug in it that misses page boundary crossings.

DON'T USE IT!

The newer 65C02's have fixed this bug, but they are not yet in wide use at this writing.

We will return to the main code by the exact opposite way we got into IMPRINT. First we shove the high half of the return address minus one, or STRP2+1 onto the stack, and then we shove the low half of the return address minus one, or STRP2, onto the stack. Miraculously, we are now back into a subroutine. To exit, you simply do a RTS.

On the subroutine return, you return to your mainstream code, exactly on the first valid instruction *following* your text message. Very nicely, all the text went out by way of IMPRINT, and the 6502 is ready to continue on the first valid instruction that follows the message.

Note carefully what happened. We go merrily along doing the usual op codes in the usual way. Then we JSR to some very special code that reads and then outputs everything that follows as text. This continues until an end marker. Then, the special code automatically "skips over" the text part, letting you pick back up on the conventional op codes that follow.

At no time does the 6502 see anything but legal op codes. While there is a big "hole" in your source code that holds text, this part of your source code never gets to the CPU. Nifty.

A Demo

To use IMPRINT, just load it into a known location in your Apple. In any module where you want to output a text message, insert a JSR IMPRINT, followed by the message, followed by a double zero marker. Then pick up your continuing code, just like you normally would.

DEMO2 shows us how it's done. We first initialize to the text mode, clear the screen, do a tab to center a title, and then switch to inverse. Next, our first message is put down by JSRing to IMPRINT, followed by the "Imbedded String Method" title. We then go back to normal text for a few lines, followed by an inverse "JSR," and more normal text. The messages can be combined end on end as shown. This lets you have long messages that will still print neatly on your source code listing. Once again, the assembler enters the character strings with an ASC command, and enters control commands and end markers using DFBs.

Lines 155 and 156 show how to put a prompt into a fancy cue box. You might want to modify this slightly for best IIe results.

As with the file printer, a DOS command is done by starting with a CR and EOT, or [D] followed by a legal DOS instruction. The user can pick an "E" for exit or a "C" for catalog to demonstrate DOS access. Any other key reprints the message, giving a subtle, obvious, and non-obnoxious cue to the user that he is not paying attention.

The imbedded string method is far better than the file based text printer and the previous ripoff module, particularly when lots of fixed and fairly short messages are spread out in a mix-and-match fashion from program module to module.

Elegant simplicity.

MIND BENDERS

—Show how the IMPRINT method can be used with changing, calculated, or disk-based text.

—What else can you do with the concept of a JSR, followed by parameters or values needed by that sub, imbedded in mainstream code?

—Are there any advantages to using BRK to call IMPRINT? How would you do this? What are the limitations?

—Show how "new" EDASM's byte-counting LST pseudo-op can improve this module.

—How can you link an assembler with a word processor so that long text messages can be easily edited and entered into source code?

—Under what circumstances would you NOT want to use IMPRINT?

```
PROGRAM RM-2
    IMBEDDED STRING PRINTER
```

```
----- NEXT OBJECT FILE NAME IS IMPRINT
6500:              3          ORG  $6500      ; PUT MODULE #2 AT $6500


6500:              5 ;  ***********************************************
6500:              6 ;  *                                             *
6500:              7 ;  *              -< IMPRINT MODULE >-            *
6500:              8 ;  *                                             *
6500:              9 ;  *          (IMBEDDED STRING PRINTER)          *
6500:             10 ;  *                                             *
6500:             11 ;  *       VERSION 1.0   ($6500-$66A1)           *
6500:             12 ;  *                                             *
6500:             13 ;  *                 6-15-83                     *
6500:             14 ;  *.........................................*
6500:             15 ;  *                                             *
6500:             16 ;  *             COPYRIGHT C 1983 BY             *
6500:             17 ;  *                                             *
6500:             18 ;  *       DON LANCASTER AND SYNERGETICS         *
6500:             19 ;  *       BOX 1300, THATCHER AZ., 85552         *
6500:             20 ;  *                                             *
6500:             21 ;  *       ALL COMMERCIAL RIGHTS RESERVED        *
6500:             22 ;  *                                             *
6500:             23 ;  ***********************************************


6500:             25 ;              *** WHAT IT DOES ***

6500:             27 ;  THIS MODULE OUTPUTS TEXT STRINGS OR DOS COMMANDS
6500:             28 ;  TO THE APPLE II'S OUTPUT HOOKS, USING STRINGS
6500:             29 ;  THAT ARE DIRECTLY IMBEDDED IN THE SOURCE CODE.
6500:             30 ;
6500:             31 ;
6500:             32 ;


6500:             34 ;              *** HOW TO USE IT ***

6500:             36 ;  YOUR CALLING CODE SHOULD HAVE A JSR TO IMPRINT.
6500:             37 ;  THIS JSR SHOULD BE IMMEDIATELY FOLLOWED BY AN ASCII
6500:             38 ;  STRING ENDING WITH AN $00 MARKER.
6500:             39 ;
6500:             40 ;
6500:             41 ;
```

PROGRAM RM-2, CONT'D . . .

```
6500:          44 ;                *** GOTCHAS ***

6500:          46 ;   THIS METHOD IS BEST USED FOR SHORT, UNRELATED MESSAGES
6500:          47 ;   INTERNAL TO YOUR PROGRAM.
6500:          48 ;
6500:          49 ;   MESSAGES CAN BE ANY LENGTH, BUT MORE THAN 40 CHARACTERS
6500:          50 ;   WILL NOT PRINT CLEANLY ON THE ASSEMBLY LISTING.
6500:          51 ;


6500:          53 ;              *** ENHANCEMENTS ***

6500:          55 ;   DOS COMMANDS ARE OUTPUT BY STARTING THE STRING
6500:          56 ;   WITH A CARRIAGE RETURN AND <CTRL> D.
6500:          57 ;
6500:          58 ;   TO GO DIRECTLY TO THE SCREEN, USE COUT1 RATHER THAN COUT.
6500:          59 ;   THIS IS FASTER, BUT CANNOT CONTROL DOS OR BE PRINTED.
6500:          60 ;


6500:          62 ;              *** RANDOM COMMENTS ***

6500:          64 ;   TO RUN THE DEMO, USE $6500G OR CALL 25856.
6500:          65 ;
6500:          66 ;
6500:          67 ;
6500:          68 ;
6500:          69 ;
```

PROGRAM RM-2, CONT'D . . .

```
6500:          72 ;              *** HOOKS ***


FDED:          74 COUT    EQU   $FDED    ; OUTPUT CHARACTER VIA HOOKS
FC58:          75 HOME    EQU   $FC58    ; CLEAR SCREEN
C010:          76 KBDSTR  EQU   $C010    ; KEYBOARD RESET
FB2F:          77 INIT    EQU   $FB2F    ; INITIALIZE TEXT SCREEN
FD1B:          78 KEYIN   EQU   $FD1B    ; READ KEYBOARD
F94A:          79 PRBL2   EQU   $F94A    ; PRINT X BLANKS
FE80:          80 SETINV  EQU   $FE80    ; SET INVERSE SCREEN
FE84:          81 SETNORM EQU   $FE84    ; SET NORMAL SCREEN
FCA8:          82 WAIT    EQU   $FCA8    ; TIME DELAY SET BY ACCUMULATOR


00EB:          84 STRP2   EQU   $EB      ; POINTER TO ASCII STRING




6500:          86 ;              *** TEXTFILE COMMANDS ***


0088:          88 B       EQU   $88      ; BACKSPACE
008D:          89 C       EQU   $8D      ; CARRIAGE RETURN
0084:          90 D       EQU   $84      ; DOS ATTENTION
008A:          91 L       EQU   $8A      ; LINEFEED
0060:          92 P       EQU   $60      ; FLASHING PROMPT
0000:          93 X       EQU   $00      ; END OF MESSAGE
```

PROGRAM RM-2, CONT'D . . .

```
6500:              96 ;              *** DEMO ***
6500:              97 ;
6500:              98 ;

6500:             100 ;       THE DEMO USES THE IMPRINT MODULE TO OUTPUT
6500:             101 ;       SCREEN MESSAGES AND A DOS CATALOG COMMAND.
6500:             102 ;
6500:             103 ;
6500:             104 ;
6500:             105 ;

6500:20 2F FB    107 DEMO2   JSR  INIT      ; GO TO TEXT MODE
6503:20 58 FC    108         JSR  HOME      ; CLEAR SCREEN
6506:A2 07       109         LDX  #07       ; ADD BLANKS TO START
6508:20 4A F9    110         JSR  PRBL2     ;
650B:20 80 FE    111         JSR  SETINV    ; INVERSE HEADER
650E:20 6B 66    112         JSR  IMPRINT   ; PUT DOWN HEADER

6511:8A 8A 8A    114         DFB            L,L,L
6514:C9 CD C2    115         ASC            "IMBEDDED STRING METHOD"
6517:C5 C4 C4
651A:C5 C4 A0
651D:D3 D4 D2
6520:C9 CE C7
6523:A0 CD C5
6526:D4 C8 CF
6529:C4
652A:8D 8D 00    116         DFB            C,C,X

652D:20 84 FE    118         JSR  SETNORM   ; NORMAL TEXT
6530:20 6B 66    119         JSR  IMPRINT   ; TOP TEXT LINE

6533:D7 C9 D4    121         ASC            "WITH THIS METHOD, EACH MESSAGE STRING
6536:C8 A0 D4
6539:C8 C9 D3
653C:A0 CD C5
653F:D4 C8 CF
6542:C4 AC A0
6545:C5 C1 C3
6548:C8 A0 CD
654B:C5 D3 D3
654E:C1 C7 C5
6551:A0 D3 D4
6554:D2 C9 CE
6557:C7 A0
6559:8D         122         DFB            C
```

PROGRAM RM-2, CONT'D . . .

```
655A:C6 CF CC   125        ASC              "FOLLOWS ITS OWN
655D:CC CF D7
6560:D3 A0 C9
6563:D4 D3 A0
6566:CF D7 CE
6569:A0
656A:00         126        DFB              X

656B:20 80 FE   128        JSR    SETINV    ; INVERSE TEXT
656E:20 6B 66   129        JSR    IMPRINT   ;

6571:CA D3 D2   131        ASC              "JSR"
6574:00         132        DFB              X

6575:20 84 FE   134        JSR    SETNORM   ; RETURN TO NORMAL TEXT
6578:20 6B 66   135        JSR    IMPRINT   ;   AFTER JSR

657B:A0 C3 C1   137        ASC              " CALL, IMBEDDED IN
657E:CC CC AC
6581:A0 C9 CD
6584:C2 C5 C4
6587:C4 C5 C4
658A:A0 C9 CE
658D:8D         138        DFB              C

658E:C9 D4 D3   140        ASC              "ITS OWN SOURCE CODE.  "
6591:A0 CF D7
6594:CE A0 D3
6597:CF D5 D2
659A:C3 C5 A0
659D:C3 CF C4
65A0:C5 AE A0
65A3:A0

65A4:CE CF A0   142        ASC              "NO POINTERS AND
65A7:D0 CF C9
65AA:CE D4 C5
65AD:D2 D3 A0
65B0:C1 CE C4
65B3:8D         143        DFB              C

65B4:CE CF A0   145        ASC              "NO MASTER FILE ARE NEEDED.
65B7:CD C1 D3
65BA:D4 C5 D2
65BD:A0 C6 C9
65C0:CC C5 A0
65C3:C1 D2 C5
65C6:A0 CE C5
65C9:C5 C4 C5
65CC:C4 AE
65CE:8D 8D      146        DFB              C,C
```

PROGRAM RM-2, CONT'D . . .

```
65D0:C2 C5 D3   149        ASC        "BEST USE IS FOR FIXED, SHORT MESSAGES.
65D3:D4 A0 D5
65D6:D3 C5 A0
65D9:C9 D3 A0
65DC:C6 CF D2
65DF:A0 C6 C9
65E2:D8 C5 C4
65E5:AC A0 D3
65E8:C8 CF D2
65EB:D4 A0 CD
65EE:C5 D3 D3
65F1:C1 C7 C5
65F4:D3 AE
65F6:8D 8D      150        DFB        C,C

65F8:D4 D9 D0   152        ASC        /TYPE "C" FOR CATALOG, OR "E" FOR EXIT.
65FB:C5 A0 A2
65FE:C3 A2 A0
6601:C6 CF D2
6604:A0 C3 C1
6607:D4 C1 CC
660A:CF C7 AC
660D:A0 CF D2
6610:A0 A2 C5
6613:A2 A0 C6
6616:CF D2 A0
6619:C5 D8 C9
661C:D4 AE
661E:8D 8D      153        DFB        C,C

6620:A0 A0 A0   155        ASC        "            -< >-"
6623:A0 A0 A0
6626:A0 A0 A0
6629:A0 A0 A0
662C:A0 A0 AD
662F:BC A0 BE
6632:AD
6633:88 88 88   156        DFB        B,B,B,P,B,X
6636:60 88 00
```

PROGRAM RM-2, CONT'D . . .

```
6639:2C 10 C0   159 AGAIN2  BIT   KBDSTR    ; RESET KEY STROBE
663C:20 1B FD   160 KBD2    JSR   KEYIN     ; READ KEYBOARD
663F:C9 C5      161         CMP   #$C5      ; AN "E" FOR EXIT?
6641:F0 21      162         BEQ   EXIT2     ; YES, EXIT
6643:C9 C3      163         CMP   #$C3      ; A "C" FOR CATALOG?
6645:D0 1A      164         BNE   RETRY2    ; NO, REPRINT SCREEN

6647:20 58 FC   166         JSR   HOME      ; CLEAR SCREEN FOR CATALOG
664A:20 6B 66   167         JSR   IMPRINT   ;

664D:8D 84      169         DFB   C,D       ; DOS HEADER
664F:C3 C1 D4   170         ASC   "CATALOG" ;
6652:C1 CC CF
6655:C7
6656:8D 00      171         DFB   C,X       ; DOS TRAILER

6658:20 6B 66   173         JSR   IMPRINT   ; PROMPT AFTER CATALOG
665B:8D 60 00   174         DFB             C,P,X


665E:18         176         CLC             ; BRANCH ALWAYS
665F:90 D8      177         BCC   AGAIN2    ;
6661:4C 00 65   178 RETRY2  JMP   DEMO2     ; TOO FAR FOR BRANCH
6664:20 58 FC   179 EXIT2   JSR   HOME      ; CLEAR SCREEN
6667:2C 10 C0   180         BIT   KBDSTR    ; RESET KEYSTROBE
666A:60         181         RTS             ;  AND RETURN
```

PROGRAM RM-2, CONT'D . . .

```
666B:          184 ;              *** IMPRINT MODULE ***
666B:          185 ;
666B:          186 ;


666B:          188 ;              THIS MODULE UNPOPS THE STACK TO FIND THE
666B:          189 ;              IMBEDDED STRING.  IT OUTPUTS ONE CHARACTER
666B:          190 ;              AT A TIME TILL A $00 MARKER IS FOUND. THEN
666B:          191 ;              IT JUMPS BACK TO THE CALLING PROGRAM JUST
666B:          192 ;              BEYOND THE STRING.
666B:          193 ;


666B:8E A0 66  195 IMPRINT STX XSAV2      ; SAVE REGISTERS
666E:8C A1 66  196         STY YSAV2      ;
6671:8D 9F 66  197         STA ASAV2      ;

6674:68        199         PLA            ; GET POINTER LOW AND SAVE
6675:85 EB     200         STA STRP2      ;
6677:68        201         PLA            ; GET POINTER HIGH AND SAVE
6678:85 EC     202         STA STRP2+1    ;

667A:A0 00     204         LDY #$00       ; NO INDEXING
667C:E6 EB     205 NXTCHR2 INC STRP2      ; GET NEXT HIGH ADDRESS
667E:D0 02     206         BNE NOC2       ; SKIP IF NO CARRY
6680:E6 EC     207         INC STRP2+1    ; INCREMENT HIGH ADDRESS
6682:B1 EB     208 NOC2    LDA (STRP2),Y  ; GET CHARACTER
6684:F0 09     209         BEQ END2       ; IF ZERO MARKER
6686:20 9E 66  210         JSR HOOK2      ; FOR SPECIAL EFFECTS
6689:20 ED FD  211         JSR COUT       ; PRINT CHARACTER
668C:18        212         CLC            ; BRANCH ALWAYS
668D:90 ED     213         BCC NXTCHR2    ;

668F:A5 EC     215 END2    LDA STRP2+1    ; RESTORE PC LOW
6691:48        216         PHA            ;
6692:A5 EB     217         LDA STRP2      ; RESTORE PC HIGH
6694:48        218         PHA            ;
6695:AE A0 66  219         LDX XSAV2      ;
6698:AC A1 66  220         LDY YSAV2      ; RESTORE REGISTERS
669B:AD 9F 66  221         LDA ASAV2      ;
669E:60        222 HOOK2   RTS            ; AND EXIT



669F:          224 ;              *** STASH ***


669F:00        226 ASAV2   DFB $00        ; ACCUMULATOR SAVE
66A0:00        227 XSAV2   DFB $00        ; X-REGISTER SAVE
66A1:00        228 YSAV2   DFB $00        ; Y-REGISTER SAVE

*** SUCCESSFUL ASSEMBLY: NO ERRORS
```

MONITOR TIME DELAY

**how to use a monitor subrou-
tine for sounds, animation, and
other timing**

If everyone is always worried about getting their programs to run
fast enough, why on earth would you ever *purposely* want to stall for
time?

Because, of course, some of the most useful and most interesting
Apple uses center on carefully controlled sequences of time delays.
The most obvious applications are in sound and music, where you
wait for a while, and then change the position of a speaker cone. How
long you wait sets the *pitch* of the tone, while the number of times
you change the cone sets the *duration* of the note. Much more on this
in the next two ripoff modules.

Another place where you purposely want to delay precise amounts
of time involves baud-rate generation. Most often, though, these
repeated time delays are done outside your CPU with a special serial
transmitter chip. But other times, your CPU can be asked to generate
a special frequency or a timing waveform that involves carefully con-
trolled delays.

Producing the 40-kHz ultrasonic control signal for a BSR remote
power controller is one use. Here a few bytes of software can *replace*
bunches of specialized and unneeded hardware. Many industrial uses
of Apples involve function and signal generators *of one* sort or
another.

The real biggie of the time delay world centers on animation. To animate something, you put a pattern on the screen, wait a while, and then replace or modify that pattern into something different. Done just right, the changing patterns will give you the illusion of continuous motion. One very new use of Apple timing lets you carefully lock your animation to your video displays. This offers you everything from flawless and glitchless animation to mixing and matching of text, HIRES, and LORES together all at once.

Finally, there are the long term uses of time delay. Things that control appliances, turn on sprinklers, or that keep hourly, daily, weekly, or even monthly tabs on whatever it is that needs its tabs kept.

As with any programming technique, there are several different popular ways you can go about stalling for time. Which one you use depends on what you are trying to accomplish and how much else has to happen while the time delay is taking place.

The fundamental unit of Apple time delay is called a *clock cycle.* One clock cycle is roughly one microsecond, so you will need around one million of these for a one second delay. The Apple clock cycles are crystal controlled, so they are themselves accurate to at least one part in a million.

But there is one possible source of inaccuracy that will get to you if you aren't careful. Apple clock cycles are *not* precisely one microsecond long . . .

An Apple clock cycle is ROUGHLY 1 microsecond long.

An Apple clock cycle takes EXACTLY 0.978 microseconds or 978 nanoseconds.

A microsecond takes up EXACTLY 1.023 Apple clock cycles.

Just to confuse you further, these times are *average* values. Each 65th clock cycle is one-seventh longer than all the rest. This is done to uniquely solve a sticky timing glitch. The result is a tiny, and usually negligible, jitter in outside-world timing applications.

For most everyday needs, you simply say a cycle is a microsecond, and live with the two percent error you get. But, if you need an exact number of Apple clock cycles, or an exactly specified time delay, you have to "fine tune" your thinking to get precisely what you need.

As examples, locking to an Apple field takes a precise delay of 17030 Apple clock cycles, no more and no less. The time does not matter here; the cycles are everything. If you must have precisely one second of delay, you should use 1,022,727 clock cycles and not an even million. But never make things bunches more precise than you really need, since extra accuracy is often a pointless waste of time and effort.

I guess I really get into time delay techniques whole hog, since some of the most mind-blowing and most challenging Apple uses involve carefully controlled time delays where an exact result has to be gotten in an exact number of cycles. This, of course, is what most of the cheap video stuff was all about, (Sams 21524 and 21723) and is an ongoing challenge in the Enhance series (Sams 21822, etc.).

Sometimes you will only want to delay for a few clock cycles. Other times you might need great heaping bunches of cycles. So, you have a choice of time delay methods. Here, going from short to long, are some possible . . .

WAYS TO STALL FOR TIME

---

cycle burner uppers
simple loop
monitor delay

triple monitor delay
combined use
offloading

Burning up clock cycles is one good way for short time delays of a few microseconds. What you do is throw in some Apple CPU commands that don't really do anything but burn up clock cycles. These might be used to equalize two paths through time critical code, to provide video positioning, or be used anywhere else you need only a few cycles of correction.

Here are some standard . . .

CYCLE BURNER UPPERS

---

2 cycles ... NOP
3 cycles ... BCC taken or JMP
4 cycles ... NOP and NOP

5 cycles ... NOP and BCC taken
6 cycles ... NOP and NOP and NOP
7 cycles ... PHA and PLA

The object of the game is to use as few code bytes as possible for your delay and to not hurt anything else in the way of flags or working registers. You can find the "efficiency" of a 6502 instruction by dividing the number of cycles delayed by the number of bytes needed. NOPs are often your safest bet since they do the least damage.

Doing one single cycle of delay gets tricky. While many of the "illegal" commands in the 65C02 default to single cycle NOPs, there is no obvious way to do a single cycle delay with the older and stock 6502's. The way I usually handle a single delay cycle is to set up the *difference* between two paths that have even and odd total clock cycles. For instance, if your carry flag is set, a BCC takes up two cycles and a BCS takes up three.

If you do use branches for exact time delays, watch your page boundary crossings! A mysterious "extra" clock cycle or two will sometimes result if your code crosses a page when you didn't expect it to.

Once you get good at it, you should try to build your time delays into code commands, so that your code does other *good stuff* at the same time it is providing your time delay.

Needless to say, cycle burner uppers get old for more than a few clock cycles worth of delay. There are obviously better ways to stall for a second than by using 511,350 NOPs in a row.

What usually happens for longer delays is that you try to take up *most* of the delay with some efficient code, and then, if you have to, "equalize" with cycle burner uppers to hit any magic values you need.

The next larger arrow in our delay quiver is the simple loop. Like so . . .

```
              LDX #$06
       LOOP   DEX
              BNE LOOP
```

What you have done is filled a loop with a value and then counted it down. Go through the math, and you will find you get a total of 5N+1 clock cycles. N here is the *hex* value you initially load the loop with. So this dude is good for 6, 11, 16, 21, 26, . . . clock cycles.

Note that a loop value of zero will go all the way around, rather than falling through, for a total of 1281 clock cycles. In terms of audio frequencies, this equals a square wave's half-period of just under 400 Hz. The reason the zero is missed is that it immediately is decremented to $FF and thus gets "caught" by the taken BNE branch. Zero is thus the *maximum* possible loop time.

For longer delays, you can put extra cycle burner uppers inside the loop, or else go to a loop within a loop. As examples, a NOP inside your loop changes the formula to 7N+1 cycles, while two simple loops inside each other will get you over a tenth of a second of delay.

But there is a much better way for medium-length delays. There is a super elegant and super versatile time delay built into the Apple monitor that is most useful for longer time delays. To use this routine, all you do is put a magic value into the accumulator and call the routine. Like this . . .

USING THE MONITOR TIME DELAY

---

1. Put a magic value in A.
2. Do a JSR to $FCA8.

And that's all there is to it. Go through the code on this, and you'll find it to be disgustingly elegant. All that gets used is the accumulator and two "borrowed" stack locations. Nothing else is tied up or used at all.

Part of the elegance involves the timing range you get. You can go anywhere from 13 clock cycles, on up to a sixth of a second, starting with only a single 8-bit magic value. Very conveniently, the available 256 time-delay values are spread out in a somewhat "log" fashion, so you get "tight" spacing on small delays and "wide" spacing on long delays.

The only reason this routine is not used as much as it should be is that the formula for the "magic" delay value is scary. Spooky even. And so misunderstood that even Apple has misprinted its formula in several different places.

The magic formula, expressed in *clock cycles* is . . .

$$\text{MONITOR DELAY CYCLES} = 13 + 13.5*A + 2.5*A*A$$

If you want the time delay in microseconds, just multiply the above result by 0.978.

Apple failed to do this on page 63 of the *Apple II Reference Manual* and on page 223 of the *Apple IIe Reference Manual*. To correct your manual, cross out "microseconds" and write in "clock cycles!"

For milliseconds, divide the scaled result by 1000, and for seconds of delay, divide by a million. As usual, don't forget to convert your decimal values into hex before assembling them, or your delay will end up wrong just about every time.

Since that formula is so ugly and nasty that it might even scare an eighth grader, we'll just spell it all out for you in longhand . . .

TIME DELAY VALUES FOR THE MONITOR WAIT SUBROUTINE

HEX A	DECIMAL A	CYCLES	MICROSECONDS	MILLISECONDS
$00	0	13	12	.012
$01	1	29	28	.028
$02	2	50	48	.048
$03	3	76	74	.074
$04	4	107	104	.104
$05	5	143	139	.139
$06	6	184	179	.179
$07	7	230	224	.224
$08	8	281	274	.274
$09	9	337	329	.329
$0A	10	398	389	.389
$0B	11	464	453	.453
$0C	12	535	522	.522
$0D	13	611	597	.597
$0E	14	692	676	.676
$0F	15	778	760	.76
$10	16	869	849	.849
$11	17	965	943	.943
$12	18	1066	1042	1.042
$13	19	1172	1145	1.145
$14	20	1283	1254	1.254
$15	21	1399	1367	1.367
$16	22	1520	1485	1.485
$17	23	1646	1608	1.608
$18	24	1777	1737	1.737
$19	25	1913	1869	1.869
$1A	26	2054	2007	2.007
$1B	27	2200	2150	2.15
$1C	28	2351	2298	2.298
$1D	29	2507	2450	2.45
$1E	30	2668	2608	2.608
$1F	31	2834	2770	2.77

TIME DELAY TABLE, CONTINUED				
HEX A	DECIMAL A	CYCLES	MICROSECONDS	MILLISECONDS
$20	32	3005	2937	2.937
$21	33	3181	3109	3.109
$22	34	3362	3286	3.286
$23	35	3548	3468	3.468
$24	36	3739	3654	3.654
$25	37	3935	3846	3.846
$26	38	4136	4043	4.043
$27	39	4342	4244	4.244
$28	40	4553	4450	4.45
$29	41	4769	4661	4.661
$2A	42	4990	4877	4.877
$2B	43	5216	5098	5.098
$2C	44	5447	5324	5.324
$2D	45	5683	5555	5.555
$2E	46	5924	5790	5.79
$2F	47	6170	6031	6.031
$30	48	6421	6276	6.276
$31	49	6677	6526	6.526
$32	50	6938	6782	6.782
$33	51	7204	7042	7.042
$34	52	7475	7306	7.306
$35	53	7751	7576	7.576
$36	54	8032	7851	7.851
$37	55	8318	8130	8.13
$38	56	8609	8415	8.415
$39	57	8905	8704	8.704
$3A	58	9206	8999	8.999
$3B	59	9512	9298	9.298
$3C	60	9823	9602	9.602
$3D	61	10139	9911	9.911
$3E	62	10460	10224	10.224
$3F	63	10786	10543	10.543
$40	64	11117	10867	10.867
$41	65	11453	11195	11.195
$42	66	11794	11528	11.528
$43	67	12140	11867	11.867
$44	68	12491	12210	12.21
$45	69	12847	12558	12.558
$46	70	13208	12911	12.911
$47	71	13574	13268	13.268
$48	72	13945	13631	13.631
$49	73	14321	13999	13.999
$4A	74	14702	14371	14.371
$4B	75	15088	14748	14.748
$4C	76	15479	15130	15.13
$4D	77	15875	15518	15.518
$4E	78	16276	15910	15.91
$4F	79	16682	16306	16.306

TIME DELAY TABLE, CONTINUED				
HEX A	DECIMAL A	CYCLES	MICROSECONDS	MILLISECONDS
$50	80	17093	16708	16.708
$51	81	17509	17115	17.115
$52	82	17930	17526	17.526
$53	83	18356	17943	17.943
$54	84	18787	18364	18.364
$55	85	19223	18790	18.79
$56	86	19664	19221	19.221
$57	87	20110	19657	19.657
$58	88	20561	20098	20.098
$59	89	21017	20544	20.544
$5A	90	21478	20995	20.995
$5B	91	21944	21450	21.45
$5C	92	22415	21911	21.911
$5D	93	22891	22376	22.376
$5E	94	23372	22846	22.846
$5F	95	23858	23321	23.321
$60	96	24349	23801	23.801
$61	97	24845	24286	24.286
$62	98	25346	24776	24.776
$63	99	25852	25270	25.27
$64	100	26363	25770	25.77
$65	101	26879	26274	26.274
$66	102	27400	26783	26.783
$67	103	27926	27298	27.298
$68	104	28457	27817	27.817
$69	105	28993	28341	28.341
$6A	106	29534	28869	28.869
$6B	107	30080	29403	29.403
$6C	108	30631	29942	29.942
$6D	109	31187	30485	30.485
$6E	110	31748	31034	31.034
$6F	111	32314	31587	31.587
$70	112	32885	32145	32.145
$71	113	33461	32708	32.708
$72	114	34042	33276	33.276
$73	115	34628	33849	33.849
$74	116	35219	34427	34.427
$75	117	35815	35009	35.009
$76	118	36416	35597	35.597
$77	119	37022	36189	36.189
$78	120	37633	36786	36.786
$79	121	38249	37389	37.389
$7A	122	38870	37996	37.996
$7B	123	39496	38608	38.608
$7C	124	40127	39224	39.224
$7D	125	40763	39846	39.846
$7E	126	41404	40473	40.473
$7F	127	42050	41104	41.104

TIME DELAY TABLE, CONTINUED

HEX A	DECIMAL A	CYCLES	MICROSECONDS	MILLISECONDS
$80	128	42701	41740	41.74
$81	129	43357	42382	42.382
$82	130	44018	43028	43.028
$83	131	44684	43679	43.679
$84	132	45355	44335	44.335
$85	133	46031	44996	44.996
$86	134	46712	45661	45.661
$87	135	47398	46332	46.332
$88	136	48089	47007	47.007
$89	137	48785	47688	47.688
$8A	138	49486	48373	48.373
$8B	139	50192	49063	49.063
$8C	140	50903	49758	49.758
$8D	141	51619	50458	50.458
$8E	142	52340	51163	51.163
$8F	143	53066	51872	51.872
$90	144	53797	52587	52.587
$91	145	54533	53306	53.306
$92	146	55274	54031	54.031
$93	147	56020	54760	54.76
$94	148	56771	55494	55.494
$95	149	57527	56233	56.233
$96	150	58288	56977	56.977
$97	151	59054	57726	57.726
$98	152	59825	58479	58.479
$99	153	60601	59238	59.238
$9A	154	61382	60001	60.001
$9B	155	62168	60770	60.77
$9C	156	62959	61543	61.543
$9D	157	63755	62321	62.321
$9E	158	64556	63104	63.104
$9F	159	65362	63892	63.892
$A0	160	66173	64685	64.685
$A1	161	66989	65482	65.482
$A2	162	67810	66285	66.285
$A3	163	68636	67092	67.092
$A4	164	69467	67905	67.905
$A5	165	70303	68722	68.722
$A6	166	71144	69544	69.544
$A7	167	71990	70371	70.371
$A8	168	72841	71203	71.203
$A9	169	73697	72040	72.04
$AA	170	74558	72881	72.881
$AB	171	75424	73728	73.728
$AC	172	76295	74579	74.579
$AD	173	77171	75435	75.435
$AE	174	78052	76297	76.297
$AF	175	78938	77163	77.163

TIME DELAY TABLE, CONTINUED				
HEX A	DECIMAL A	CYCLES	MICROSECONDS	MILLISECONDS
$B0	176	79829	78034	78.034
$B1	177	80725	78910	78.91
$B2	178	81626	79790	79.79
$B3	179	82532	80676	80.676
$B4	180	83443	81566	81.566
$B5	181	84359	82462	82.462
$B6	182	85280	83362	83.362
$B7	183	86206	84267	84.267
$B8	184	87137	85177	85.177
$B9	185	88073	86092	86.092
$BA	186	89014	87012	87.012
$BB	187	89960	87937	87.937
$BC	188	90911	88867	88.867
$BD	189	91867	89801	89.801
$BE	190	92828	90740	90.74
$BF	191	93794	91685	91.685
$C0	192	94765	92634	92.634
$C1	193	95741	93588	93.588
$C2	194	96722	94547	94.547
$C3	195	97708	95511	95.511
$C4	196	98699	96479	96.479
$C5	197	99695	97453	97.453
$C6	198	100696	98432	98.432
$C7	199	101702	99415	99.415
$C8	200	102713	100403	100.403
$C9	201	103729	101396	101.396
$CA	202	104750	102394	102.394
$CB	203	105776	103397	103.397
$CC	204	106807	104405	104.405
$CD	205	107843	105418	105.418
$CE	206	108884	106435	106.435
$CF	207	109930	107458	107.458
$D0	208	110981	108485	108.485
$D1	209	112037	109518	109.518
$D2	210	113098	110555	110.555
$D3	211	114164	111597	111.597
$D4	212	115235	112644	112.644
$D5	213	116311	113695	113.695
$D6	214	117392	114752	114.752
$D7	215	118478	115814	115.814
$D8	216	119569	116880	116.88
$D9	217	120665	117952	117.952
$DA	218	121766	119028	119.028
$DB	219	122872	120109	120.109
$DC	220	123983	121195	121.195
$DD	221	125099	122286	122.286
$DE	222	126220	123382	123.382
$DF	223	127346	124482	124.482

	TIME DELAY TABLE, CONTINUED			
HEX A	DECIMAL A	CYCLES	MICROSECONDS	MILLISECONDS
$E0	224	128477	125588	125.588
$E1	225	129613	126698	126.698
$E2	226	130754	127814	127.814
$E3	227	131900	128934	128.934
$E4	228	133051	130059	130.059
$E5	229	134207	131189	131.189
$E6	230	135368	132324	132.324
$E7	231	136534	133464	133.464
$E8	232	137705	134608	134.608
$E9	233	138881	135758	135.758
$EA	234	140062	136913	136.913
$EB	235	141248	138072	138.072
$EC	236	142439	139236	139.236
$ED	237	143635	140405	140.405
$EE	238	144836	141579	141.579
$EF	239	146042	142758	142.758
$F0	240	147253	143942	143.942
$F1	241	148469	145130	145.13
$F2	242	149690	146324	146.324
$F3	243	150916	147522	147.522
$F4	244	152147	148726	148.726
$F5	245	153383	149934	149.934
$F6	246	154624	151147	151.147
$F7	247	155870	152365	152.365
$F8	248	157121	153588	153.588
$F9	249	158377	154816	154.816
$FA	250	159638	156048	156.048
$FB	251	160904	157286	157.286
$FC	252	162175	158528	158.528
$FD	253	163451	159776	159.776
$FE	254	164732	161028	161.028
$FF	255	166018	162285	162.285

A copy of this listing appears on the companion diskette as a bonus program. Make as many copies as you like in any format you care to.

We'll find out just how to use the monitor delay subroutine shortly. Note that you do not get *every* value in the range you need. What you do is take the nearest value and then either live with it or else "pad" it with cycle burner uppers.

Let's quickly round out our survey of ways to stall for time. If you use the monitor delay three times in a row with just the right *different* "magic" values, you can hit practically any exact value over a one to three hundred millisecond range. This was needed and used extensively in *Enhancing Your Apple II* (Sams 21822).

As another bonus program on the support diskette for this book, we'll throw in an automatic magic number finder that quickly solves the triple delay problem for you. The task is not trivial. More details on this support diskette are found inside the back cover.

On longer delays, it is always best to try and do other things while you are stalling for time. For instance, you can increment a random number pair while you are waiting for someone to press a key. Or you can use your animated graphics plotting time as part of the time delay for a sound. Always suspect long times spent "wheel spinning," and see if you can't replace stalling code with some useful yet time consuming task instead . . .

Avoid "wheel spinning" for wheel spinning's sake.

ALWAYS try and make your time delay code handle other useful tasks.

The "best" way to stall for time is to have something other than the CPU do the delaying for you. This frees up your Apple to go on to do other useful things. For instance, you can send a single and fast "transmit" command to a serial card whose separate UART takes its good old time outputting a serial code. Or, send your music commands to a music chip. Or your timer commands to a timer chip. Or use a real time clock chip to interrupt your Apple for those things that take really long time delays, such as control of a sprinkler system.

Unfortunately, all of these "offloaders" take special hardware and add to your system cost. They also limit who you can sell your product to. Sometimes it is best to do your initial timing with the CPU and then later offload cumbersome timing once your product is better defined.

Using the Monitor Delay

Let's find out how to use the monitor delay for some exciting and noisy animation. So stunning, in fact, that it might earn a fifth grader a B— if his teacher was feeling generous. While we are at it, we will pick up some fundamentals of LORES plotting using the existing monitor LORES subs.

Many people look down on LORES, but a thorough understanding of LORES graphics is almost essential if you are ever going to handle HIRES. The IIe now offers double LORES graphics of 24 × 80 color blocks, which considerably eases the "chunkiness" of the display.

LORES animation and repeated mapping can be done much faster and with far fewer bytes than can be done in HIRES. And, thanks to the exact field sync of the Enhancing series, you can easily mix and match text, LORES, and HIRES together anyplace you want on the screen all at the same time.

Our main program is called DEMO3. DEMO3 consists of three subroutines, just as any "high level" code should be made up entirely of subroutine calls. The first subroutine clears the screen and draws an empty bucket on the screen. The second subroutine fills the bucket at a one layer per second rate. The third subroutine causes an explosion when the bucket is completely filled. Calling the fire department or pressing any key ends the explosion.

We will let you do your own flowchart on this, since nothing sneaky is involved.

The first subroutine is called DRAWCUP. This one initializes the LORES screen and clears it using the existing SETGR and CLRSCR monitor subs. You then set the bucket color to green using the SETCOL subroutine, and then draw your bucket.

Bucket drawing is done using the HLINE and VLINE monitor subroutines. You enter HLINE with the vertical position in the accumulator, the left end line position in the Y register, and the right end line position in page zero location $2C.

Alike but different somehow, you enter VLINE with the horizontal position in the Y register, the top-most line position in the accumulator, and the bottom-most line position in page zero location $2D.

Note how the use of labels HEND for $2C and VBOT for $2D eases remembering these values.

The FILLCUP subroutine fills the cup one level at a time, spending one second per level. Several sub-subs are involved. The TENTHS subroutine uses the monitor delay to produce one-tenth of a second delay. In this demo, we won't worry about exact timing values, since they are not at all critical.

Since we cannot do a one-second delay directly with the monitor sub, we instead use our own SECONDS subroutine, which calls the TENTHS subroutine ten times in a row to get a one-second delay.

To round out our time delays, there is also a TENMSEC subroutine that generates a 10-millisecond delay, useful to produce a sound effect as part of the BRACK subroutine. More details on sound effects appear in the next two ripoff modules.

The "explosion" is done by rapidly changing the screen modes while whapping the speaker. It sounds and looks awful.

MIND BENDERS

—Why does the liquid stay inside the cup, rather than overwriting the existing cup sides?

—What are the *exact* time delays in use, including all sub timing and all overhead code?

—Improve the animation and the display so it would earn a seventh grader an A—.

—Only certain cup and liquid colors are compatible on an average color set. Why? Which combinations look best in both color and black and white?

—Redo this demo in HIRES. Do an on-screen splash. Then include a *real* squirt gun in your demo.

PROGRAM RM-3
MONITOR TIME DELAY

```
----- NEXT OBJECT FILE NAME IS TIME DELAY
6700:           3              ORG  $6700      ; PUT MODULE #3 AT $6700


6700:           5 ;    ******************************************
6700:           6 ;    *                                        *
6700:           7 ;    *            -< TIME DELAY >-             *
6700:           8 ;    *                                        *
6700:           9 ;    *          (USING MONITOR WAIT)          *
6700:          10 ;    *                                        *
6700:          11 ;    *    VERSION 1.0   ($6700-$67AC)         *
6700:          12 ;    *                                        *
6700:          13 ;    *             11-24-82                   *
6700:          14 ;    *...................................... *
6700:          15 ;    *                                        *
6700:          16 ;    *          COPYRIGHT C 1982 BY           *
6700:          17 ;    *                                        *
6700:          18 ;    *    DON LANCASTER AND SYNERGETICS       *
6700:          19 ;    *    BOX 1300, THATCHER AZ., 85552       *
6700:          20 ;    *                                        *
6700:          21 ;    *    ALL COMMERCIAL RIGHTS RESERVED      *
6700:          22 ;    *                                        *
6700:          23 ;    ******************************************


6700:          25 ;          *** WHAT IT DOES ***

6700:          27 ;    THIS PROGRAM SHOWS HOW TO USE THE MONITOR WAIT
6700:          28 ;    SUBROUTINE FOR TIME DELAYS OF 0.01, 0.1, 1.0,
6700:          29 ;    AND 10.0 SECONDS.
6700:          30 ;
6700:          31 ;
6700:          32 ;


6700:          34 ;          *** HOW TO USE IT ***

6700:          36 ;    TO USE, RUN THE DEMO BY $6700G FROM MACHINE LANGUAGE
6700:          37 ;    OR CALL 26368 FROM APPLESOFT.
6700:          38 ;
6700:          39 ;    THEN ADAPT THE METHOD AND RESULTS TO YOUR OWN
6700:          40 ;    NEEDS.
6700:          41 ;
```

PROGRAM RM-3, CONT'D . . .

```
6700:          44 ;                    *** GOTCHAS ***

6700:          46 ;    THE ACCUMULATOR IS DESTROYED BY THE WAIT SUBROUTINE.
6700:          47 ;
6700:          48 ;    MACHINE TIME AND PEOPLE TIME DIFFER! ONE CLOCK CYCLE
6700:          49 ;    EQUALS 0.976 MICROSECONDS, AND NOT 1.000 MICROSECONDS!
6700:          50 ;
6700:          51 ;    THIS SLIGHT DIFFERENCE CAN SOMETIMES BE SIGNIFICANT.


6700:          53 ;                  *** ENHANCEMENTS ***

6700:          55 ;    DEMO3 ALSO SHOWS YOU SEVERAL TRICKS INVOLVED WHEN
6700:          56 ;    YOU USE THE LORES SCREEN.
6700:          57 ;
6700:          58 ;
6700:          59 ;
6700:          60 ;


6700:          62 ;                *** RANDOM COMMENTS ***

6700:          64 ;    IF YOU NEED AN EXACT NUMBER OF MACHINE CYCLES THAT
6700:          65 ;    CANNOT BE HIT DIRECTLY WITH WAIT, TRY USING WAIT TWO
6700:          66 ;    OR THREE TIMES USING DIFFERENT A VALUES.
6700:          67 ;
6700:          68 ;
6700:          69 ;
```

PROGRAM RM-3, CONT'D . . .

```
6700:           72 ;          *** HOOKS ***

F832:           74 CLRSCR   EQU   $F832    ; CLEAR FULL LORES SCREEN
002C:           75 HEND     EQU   $2C      ; RIGHT END OF LORES H LINE
C057:           76 HIRES    EQU   $C057    ; HIRES SOFT SWITCH
F819:           77 HLINE    EQU   $F819    ; HORIZ LORES LINE
FB2F:           78 INIT     EQU   $FB2F    ; INITIALIZE TEXT SCREEN
C000:           79 IOADR    EQU   $C000    ; KEYBOARD INPUT
C010:           80 KBDSTR   EQU   $C010    ; KEYSTROBE RESET
C056:           81 LORES    EQU   $C056    ; LORES SOFT SWITCH
C053:           82 LOWSCR   EQU   $C053    ; PAGE ONE SOFT SWITCH
C052:           83 MIXCLR   EQU   $C052    ; FULL GRAPHICS SCREEN
F864:           84 SETCOL   EQU   $F864    ; SET LORES COLOR
FB40:           85 SETGR    EQU   $FB40    ; SET UP GRAPHICS SCREEN
C030:           86 SPKR     EQU   $C030    ; SPEAKER CLICK OUTPUT
C050:           87 TXTCLR   EQU   $C050    ; GRAPHICS ON SOFT SWITCH
C051:           88 TXTSET   EQU   $C051    ; TEXT ON SOFT SWITCH
002D:           89 VBOT     EQU   $2D      ; BOTTOM OF LORES V LINE
F828:           90 VLINE    EQU   $F828    ; VERTICAL LORES LINE
FCA8:           91 WAIT     EQU   $FCA8    ; TIME DELAY SET BY ACCUMULATOR
```

PROGRAM RM-3, CONT'D . . .

```
6700:           94 ;               *** DEMO ***
6700:           95 ;
6700:           96 ;


6700:           98 ;          THE DEMO FILLS A LORES BUCKET
6700:           99 ;          EACH SECOND TILL OVERFLOW,
6700:          100 ;          TICKING OFF EACH TENTH OF A SECOND.
6700:          101 ;
6700:          102 ;
6700:          103 ;


6700:20 0A 67  105 DEMO3   JSR  DRAWCUP   ; DRAW LORES CUP
6703:20 3F 67  106         JSR  FILLCUP   ; FILL CUP
6706:20 5C 67  107         JSR  EXPLODE   ; THEN EXPLODE
6709:60        108         RTS            ; AND EXIT


670A:          110 ;             *** DRAWCUP SUBROUTINE ***
670A:          111 ;
670A:          112 ;     THE DRAWCUP SUBROUTINE DRAWS A LORES CUP ON THE SCREEN.
670A:          113 ;
670A:          114 ;
670A:          115 ;


670A:20 40 FB  117 DRAWCUP JSR  SETGR     ; INIT LORES SCREEN
670D:2C 52 C0  118         BIT  MIXCLR    ; FULL SCREEN GRAPHICS
6710:20 32 F8  119         JSR  CLRSCR    ; CLEAR FULL LORES SCREEN
6713:A9 04     120         LDA  #$04      ; USE GREEN BUCKET
6715:20 64 F8  121         JSR  SETCOL    ; AND SET COLOR
6718:A9 19     122         LDA  #$19      ; DRAW BASE
671A:85 2C     123         STA  HEND      ;
671C:A0 0C     124         LDY  #$0C      ;
671E:A9 1E     125         LDA  #$1E      ;
6720:20 19 F8  126         JSR  HLINE     ; AND PLOT IT
6723:A9 1E     127         LDA  #$1E      ; DRAW SIDES
6725:85 2D     128         STA  VBOT      ;
6727:A0 0D     129         LDY  #$0D      ;
6729:A9 14     130         LDA  #$14      ;
672B:20 28 F8  131         JSR  VLINE     ; AND DRAW LEFT SIDE
672E:A9 14     132         LDA  #$14      ;
6730:A0 18     133         LDY  #$18      ;
6732:20 28 F8  134         JSR  VLINE     ; AND DRAW RIGHT SIDE
6735:A9 06     135         LDA  #$06      ; SET COLOR FOR FILL
6737:20 64 F8  136         JSR  SETCOL    ;
673A:C6 2C     137         DEC  HEND      ; FILL INSIDE RIGHT
673C:C6 2C     138         DEC  HEND
673E:60        139         RTS            ; AND RETURN
```

PROGRAM RM-3, CONT'D . . .

```
673F:              142 ;      *** FILLCUP SUBROUTINE ***
673F:              143 ;
673F:              144 ;   THIS SUBROUTINE FILLS THE CUP AT
673F:              145 ;   A ONE SECOND PER LEVEL RATE.
673F:              146 ;
673F:              147 ;


673F:A9 0A         149 FILLCUP LDA   #$0A       ; FOR TEN TRIPS
6741:8D AC 67      150         STA   CUPHI      ;  SAVE INDEX
6744:20 53 67      151 AGAIN3  JSR   SECONDS    ; DELAY VIA SECONDS SUB
6747:20 A0 67      152         JSR   POUR       ; ADD TO LEVEL
674A:20 86 67      153         JSR   BRACK3     ; MAKE NOISE
674D:CE AC 67      154         DEC   CUPHI      ; NEXT CUP LEVEL
6750:D0 F2         155         BNE   AGAIN3
6752:60            156         RTS              ; AND EXIT


6753:              158 ;      *** SECONDS SUBROUTINE ***

6753:              160 ;
6753:              161 ;
6753:              162 ;
6753:              163 ;


6753:A0 0A         165 SECONDS LDY   #$0A       ; FOR TEN TENTHS
6755:20 94 67      166 NEXT3   JSR   TENTHS     ; DELAY FOR A TENTH
6758:88            167         DEY              ;
6759:D0 FA         168         BNE   NEXT3      ; REPEAT TILL DONE
675B:60            169         RTS              ; THEN EXIT
```

PROGRAM RM-3, CONT'D . . .

```
675C:              172 ;     *** EXPLODE SUBROUTINE ***


675C:2C 57 C0      174 EXPLODE BIT  HIRES    ;
675F:20 9A 67      175         JSR  TENMSEC  ; DELAY FOR TEN  MILLISECONDS
6762:2C 56 C0      176         BIT  LORES
6765:20 9A 67      177         JSR  TENMSEC  ; AND DELAY AGAIN
6768:2C 51 C0      178         BIT  TXTSET   ;
676B:20 9A 67      179         JSR  TENMSEC  ;
676E:2C 30 C0      180         BIT  SPKR     ;
6771:2C 50 C0      181         BIT  TXTCLR   ;
6774:20 9A 67      182         JSR  TENMSEC  ;
6777:2C 30 C0      183         BIT  SPKR     ; WHAP SPEAKER
677A:2C 00 C0      184         BIT  IOADR    ; CHECK FOR KEYPRESS
677D:10 DD         185         BPL  EXPLODE  ;
677F:2C 10 C0      186         BIT  KBDSTR   ; RESET KEYBOARD
6782:20 2F FB      187         JSR  INIT     ; BACK TO TEXT SCREEN
6785:60            188         RTS           ; POP STACK AND RETURN



6786:              190 ;     *** BRACK SUBROUTINE ***


6786:A0 06         192 BRACK3 LDY  #$06     ; SECONDS TONE
6788:A9 0C         193 NOTE3  LDA  #$0C     ;
678A:20 A8 FC      194         JSR  WAIT     ;
678D:2C 30 C0      195         BIT  SPKR     ;
6790:88            196         DEY           ;
6791:D0 F5         197         BNE  NOTE3    ;
```

```
PROGRAM RM-3; CONT'D . . .


6793:60            199           RTS              ;


6794:              201 ;      *** TENTHS SUBROUTINE ***
6794:              202 ;
6794:              203 ;   THIS SUB USES WAIT TO DELAY ONE TENTH
6794:              204 ;   OF A SECOND.


6794:A9 C7         206 TENTHS LDA  #$C7      ; FOR 99.415 MILLISECONDS
6796:20 A8 FC      207        JSR  WAIT      ; DELAY VIA WAIT SUB
6799:60            208        RTS            ; AND THEN RETURN


679A:              210 ;      *** TEN MILLISECONDS SUB ***
679A:              211 ;
679A:              212 ;   THIS SUB USES WAIT TO DELAY TEN MILLISECONDS.
679A:              213 ;


679A:A9 3D         215 TENMSEC LDA #$3D      ; FOR 99.415 MILLISECONDS
679C:20 A8 FC      216        JSR  WAIT      ; DELAY VIA WAIT SUB
679F:60            217        RTS            ; AND THEN RETURN


67A0:              219 ;   *** POUR SUBROUTINE ***
67A0:              220 ;

67A0:18            222 POUR   CLC            ; FILL CUP WITH LIQUID
67A1:A9 13         223        LDA  #$13      ; TOP OF CUP LEVEL
67A3:6D AC 67      224        ADC  CUPHI     ;   MINUS HEIGHT ALREADY
67A6:A0 0E         225        LDY  #$0E      ; LEFT SIDE SET
67A8:20 19 F8      226        JSR  HLINE     ; DRAW LEVEL
67AB:60            227        RTS            ; AND EXIT


67AC:              229 ;   *** STASH ***


67AC:0A            231 CUPHI  DFB  $0A       ; LEVEL IN CUP




*** SUCCESSFUL ASSEMBLY: NO ERRORS
```

OBNOXIOUS SOUNDS

an extremely versatile and compact wide-range sound-effects generator

First the bad news.

For a given amount of programming effort and add-on hardware, the Apple will always give you sound that is "thin" and animation that is "weak," when compared against an arcade video game. This happens inevitably because the Apple CPU has to take time out to generate its own sound and graphics, and because the color system is stuck with being more or less compatible with the NTSC ("Never The Same Color") broadcast television standard.

The good news, of course, is that for an extraordinary amount of creative programming effort, and for super creative use of extra hardware, you can use your Apple to knock the bytes out of *any* arcade video game or *any* other brand of personal computer. All it takes is lots of special effort that optimizes what you can do within the bounds of the actual limits of your Apple.

The next two ripoff modules show us two of the many different ways you can get sound into your programs. We will assume, for now, that you are going to use the built-in speaker of an unmodified Apple.

The "noisemaking" hardware of your Apple seems rather limiting at first glance. You have one small and tinny-sounding speaker. All the support hardware lets you do is shove the speaker cone all the way in,

or else pull it all the way out. You do this by "whapping" address location $C030 *once* each time you want to *change* the cone's position.

Technically, address $C030 is decoded and used to change the state of a binary divider, or flip-flop. The flip-flop is coupled to a special Darlington driver transistor. One whap pushes the cone in. The next pulls it out . . .

A BIT $C030 is the standard way of moving the Apple speaker's cone from the extreme position it is in to the other extreme position.

To get some useful sounds out of the Apple speaker, you decide how often you want to shove the cone back and forth, and carefully pick the time delay needed between shovings. For instance, if you keep a constant time between shovings, you will set the pitch of an audio square wave. The duration of the tone is decided by how long you continue the shoving process.

Believe it or not, you can easily get more than one note at once, have variable volume, do bell-like tones, handle speech, and do much, much more if you are a sneaky enough programmer. And your sound can be further "thickened" considerably just by adding a larger speaker to your Apple.

Before getting fancy, though, let's get two gotchas out of the road . . .

$C030 GOTCHAS

two whaps immediately following each other give you no sound . . .

USE BIT $C030, **NOT** STA $C030.

one isolated whap may not sound . . .

USE AT LEAST 3 WHAPS PER CLICK.

Due to a quirk in the Apple's timing, any time you write to a memory location, you address that location twice. The two addressings are separated by one microsecond. If you try to do a STA $C030, you end up shoving the speaker in and then pulling it back out again an impossibly brief time later. The cone barely moves in so short a time, and, surprise, surprise, you get no sound.

So, always BIT test your speaker location. Do not write to it, unless you want no sound.

The second quirk comes about because of an attempt to save Apple system power. There is a coupling capacitor in the path between the flip-flop and the speaker. This capacitor discharges on inactivity. Which means that the speaker cone is never held "in" for long periods of time. The dropout time is long compared to most tones, so you normally won't notice it.

There are two places where you might pick up the side effects of this power-down capacitor, and where it may cause you trouble. If you try to "click" the speaker just once, there's only a 50-50 chance you will get any sound at all. So, it takes *two* repeated commands, delayed by some audio value, to guarantee an isolated click. In real life, three or four repeated speaker motions are the minimum you will want to use, since some of the clicks will sound "leaner" than others.

The other place this gotcha appears happens when you send very low-pitched notes, or a very low-pitch "sweep" to your speaker. At some point, the frequency will jump up by an octave. This frequency doubling happens when the capacitor picks up enough charge to allow cone clicking in both directions.

Watch these two details if you ever get no sound or uneven sound out of your code.

As in all other Apple programming techniques, there are lots of different ways to get sound, and each of these ways will have a certain range of effects over which they are useful. Let's survey some . . .

WAYS TO GENERATE APPLE SOUND

clickety clack
calculated routine
red book tones

table method
duty cycling
offloading

With the *clickety clack* method, you simply move the speaker cone back and forth a few times, using a loop or some other obvious code. See the BRACK subroutine of the previous ripoff module for an example. The time between whappings sets the pitch while the total number of whappings sets the duration of your sound. If the pitch is constant, you get a "pure" tone. If the pitch changes, you get a "sweep." If both halves of each cycle are the same time duration, you get a "woodwind" style tone. If one half of each cycle is much longer than the other, you get a "string" style voicing.

One important exception to the clickety clackers. Do not *ever* use the standard "[G]" or "JSR $FF3A" beep. This tone is too grating to ever use in any reasonable program . . .

NEVER use the "standard" Apple beep anywhere in any of your programs!

ALWAYS kick sand in the face of anyone who does.

In the *calculated routine* method, you generate some code that decides when and where all the zero crossings are needed for a certain sound effect. This method is often used for sirens and *sweeps*,

tonal scales, frog croaks, phasors, and other short or weird "one-shot" sounds.

The good thing about the calculated routine method is that you can get some real serendipity going, and end up with some totally wild sounds that you wouldn't ever have thought possible otherwise. The bad scene about many calculated routines is that this is "old" code done the "old" way that may end up long and cumbersome, rather than short and general.

The OBNOXIOUS SOUNDS subroutine of this ripoff module will shortly explore this technique.

The *red book tones* method is a way to make monophonic music that is useful for playing songs in tempered musical scales. This involves a pitch and duration generator, and some file access tricks. More on this in the next module.

The *table method* looks up each speaker motion as needed, out of a long table. You can produce *any* possible sound this way. Most Apple-based speech uses the table method, and virtually any sound of most any complexity can be handled with a general and versatile enough program.

There are some tricks to using the table method. Getting the table to sound like you really want it to can be very involved and may take a long time. Finding some suitable coding that lets you put lots of sound in a short table is also a real hassle. Long or multiple effects really burn up the bytes. The obvious brute force method of storing a one each time you want the speaker to move can be substantially improved by going to some sort of "run length" encoding.

The best way to study table method sound is to steal the German vocabulary file out of *Castle Wolfenstein*. To grab this table, just follow the "tearing" method of Enhancement 3 in the *Enhancing Your Apple II*, Volume I (Sams 21822).

Ah yes. Duty Cycling.

Pushing the limits. Doing the impossible. How on earth can you get more than one tone at a time out of a speaker driver that you can only push or pull? How can you do variable volume? Sinewaves and flute-like or bell-like tones?

Its really very simple. Suppose you *extremely rapidly* move the speaker cone in and out, at an ultrasonic rate. The *average* cone position depends on the *average* duty cycle. For a sinewave, just let the average cone position describe a sinewave at the frequency you want. For bell tones, let the average position slowly "decay" to its "middle" value. For more than one note at once, just let the average position equal the sum of all the notes taken together at once.

If you get into some hairy math involving *Fourier coefficients,* you can easily handle chords and other multitone effects, with or without duty cycling. The whole trick is to, on the average, put the speaker cone where it ought to be when it ought to be there.

Duty cycling techniques are described in various issues of *Apple Assembly Line*.

Offloading consists of using something other than the Apple's speaker to make the noise. Simply going to a larger speaker or into a hi-fi will help "thicken" the sound bunches, and you can get stereo effects by using the speaker hardware for one channel and the cassette output port for the other. You can separately get four more channels out of the annunciator outputs of your game paddle connector.

But the real benefits of offloading take place when you send simple commands to a custom noise generator or music generator chip. Besides producing much richer and more flexible sounds, you now offload the Apple's CPU so it is free to go on to other things. All the Apple has to do is quickly pass a few parameters on to the music chip, rather than stalling around for the entire time it takes to produce the entire tone or tone sequence.

Both *General Instruments* and *Texas Instruments* are heavily into music and sound-effect generation chips. These are often the key circuits used in the fancier plug-in synthesizer cards and systems as well.

Time now for more details on . . .

The Calculated Routine Method

The calculated routine method is best done for single and isolated sound effects.

A phasor blast, of course, is the architypical example of this sort of thing. We'll show you a few dozen bytes of code that do the standard and classical phasor blast for you. But, by changing only two values, those same bytes can do a surprising variety of effects that sound wildly different.

These include some very pleasant and highly "brassy" prompt tones, musical glissades, some "cartoon" style sound, a geiger-counter simulation, and a few assorted and highly useful pips, ticks, and whopidoops. There's even a special effect called the *time bomb,* that lasts for minutes, and has all sorts of impractical joke possibilities.

The object of *any* sound program is to produce some speaker whappings separated by some time delays. The time delays set the time between zero crossings of the sound that the speaker is to produce. Usually these delays will range from 10 microseconds to 10 milliseconds or so. Faster than this and you are into ultrasonics that you cannot hear and that the speaker cone cannot follow. Slower than this breaks the sound down into individual and possibly annoying clicks.
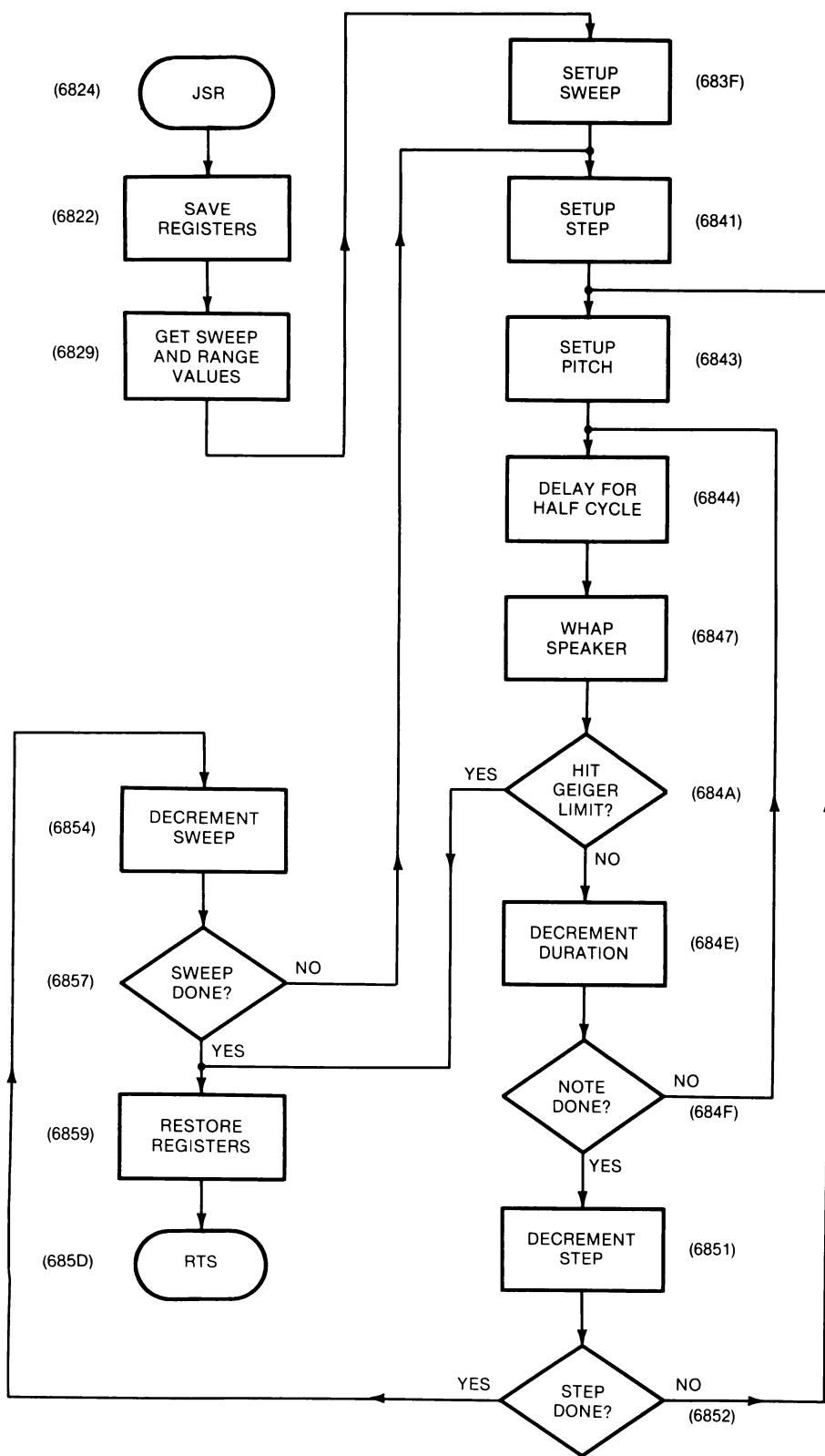
If all the time delays are the same value, you get a constant square-wave tone. The total number of time delays sets the duration of the tone, while each individual delay sets the pitch of each half-cycle of sound.

Things get interesting when you vary the time delays in a strange manner. For instance, if you make each successive time delay shorter, you get a siren or *sweep* effect that goes up in time.

The whole intent of the calculated routine method is to produce some interesting changes in the time delays that give you fat, thick, and interesting sound effects. The calculations of your routine should create a group of delay values that result in a useful sound.

Here's the flowchart for this module's calculated routine sound effects generator . . .

OBNOXIOUS SOUNDS FLOWCHART:

(6824)   JSR

(6822)   SAVE REGISTERS

(6829)   GET SWEEP AND RANGE VALUES

(683F)   SETUP SWEEP

(6841)   SETUP STEP

(6843)   SETUP PITCH

(6844)   DELAY FOR HALF CYCLE

(6847)   WHAP SPEAKER

(684A)   HIT GEIGER LIMIT?   YES   NO

(684E)   DECREMENT DURATION

(684F)   NOTE DONE?   NO   YES

(6851)   DECREMENT STEP

(6852)   STEP DONE?   YES   NO

(6854)   DECREMENT SWEEP

(6857)   SWEEP DONE?   NO   YES

(6859)   RESTORE REGISTERS

(685D)   RTS

Actually, this is nothing but a very simple sweep generator with one or two added tricks. The only two parameters under your control are how *far* you sweep and the *total number of sweeps* you use. Now,

don't go away, for you will be utterly amazed at how many *totally different* effects you can get this way. In theory, there are 65536 different effects possible. In practice, there's only two dozen or so that you will find genuinely useful and uniquely different.

Our first trick is to use the monitor delay subroutine. Remember that these delay values are "cramped together" at the short end, giving you a more or less log response. And this is just what you want for an audio sweep. A linear sweep sounds awful, since your ear is a log device that expects a few cycles change for low notes and lots of cycles change for high notes. So, the monitor delay sub automatically puts the low notes close together and the high notes far apart, just like you need.

Our second trick is to use the same value to set both the pitch and the length of each step in the sweep. This keeps things simple, yet still gives you many different sounds.

Our third trick is very sneaky. Five testing bytes are added to give you geiger counter or multiple click effects. If the sweep duration is *less* than $80, you get the complete sweep, all the way up in pitch. If the sweep duration is *greater* than $80, the sweep only goes to the $80 value and then quits.

The $80 value is extremely low in pitch. So low that you hear each cone movement as a distinct click. With the five byte code patch, values greater than $80 give you a burst of clicks. Values less than $80 give you the full sweep. So, you get two wildly and totally different classes of sound effects out of the same simple calculated routine.

We have used a sixteen-entry file to support the sound effects generator. If you only want one or two sounds, you can eliminate this file and direct poke the effects you are after. Each sound effect is specified with two values. The first decides the *number* of the sweeps produced, while the second decides *how long* each sweep is to be.

At any rate, you enter the subroutine with a number in the X register that equals the sound effect you are after. You then save all the other registers. Next you check to make sure the number is legal. If it is not, you replace it with sound effect zero. You might prefer some fancier error trapping here, but this is probably all you will really need.

Next, the sound effect number is converted into two sweep values by looking them up in the SEF effects file. The number of sweeps is grabbed first and put in an absolute location called TRPCNT4. This location will get counted down, once per each complete sweep. After this, the sweep duration is grabbed and "force fed" into the code at SWEEP4+1.

Uh, whoops. Play that one by again.

Tricks like this go by the name of *self-modifying* code. Which is legal and powerful if you know what you are doing. What you have done here is *changed* a LDY #00 command into a LDY #SWEEPS command. Note that the data value gets poked into the *second* byte of the op code! Put it anywhere else and you plow the program. Note also that any self-modifying code *must* be in RAM. EPROM need not apply.

Why?

Generally, it is safe to *pre-modify* your code like we have done here. In fact, this is a standard and powerful programming technique. Just be sure that you are changing ONLY the EXACT location you think you are. On the other hand, code that continuously changes itself on

the fly is very dangerous. Deadly even. Yet still a specialized and most useful programming technique.

Some comment . . .

> **If you self-modify code, be sure to place what you are putting EXACTLY where you intend to put it!**
>
> **One or two data values preplaced once before use is safe and standard.**
>
> **Code that continuously changes itself is often dumb and deadly.**

So much for a side trip on self-modifying code. At this point, we have a "number-of-sweeps" value in TRPCNT4, and the "length-of-the-sweep" value has been force fed into a command that loads the Y register.

Now to get sneaky. We need a third parameter. Namely, the duration value that sets the frequency for this part of our sweep. For simplicity, we just transfer Y to X, and let X set our duration and Y our pitch. For any given step of our sweep, we want all constant frequencies. Thus, we will keep Y constant while we count X down. This results in a sound that sweeps up in distinct note-like steps.

So far, so good. You transfer your pitch value to the accumulator and then use the monitor delay to stall for a half-cycle. Then, you whap the speaker. Next, you check X for the $80 value that separates the geiger effects from the long sweeps. If you have a geiger burst, you exit. For a sweep, you continue.

You continue this for X half-cycles to generate one "step" of your sweep. Then you decrement Y to go on to the next sweep step. Do this till you have completed the last step. Note that the last step is the shortest and the highest in pitch.

That should complete one sweep for you. Decrement TRPCNT4. If more sweeps are needed, then repeat the process for as many sweeps as you want. Finally, restore all the registers and exit.

There is an "oldfangled" classic cell animation demo on the companion diskette named ENGINE that you simply will not believe the first time you see and hear it. ENGINE uses the obnoxious sounds subroutine. It also has two secret ingredients called David W. Meyer, Sr., and David W. Meyer, Jr. Who, together, form one of the most fantastic father and son Apple animation teams I've ever run across anywhere, ever. And, yes, they do custom work. See the Appendix for an address.

We'll show you a simpler demo of the obnoxious sounds here and now. DEMO4 just goes through all sixteen of the sounds in order and gives you a time delay between effects.

DEMO4 produces an earth-shattering explosion a second or so after the time bomb countdown is complete. Be sure to remove all china, Ming vases, etc. from a thousand-foot radius of your Apple before running this demo.

MIND BENDERS

—Change the code so you sweep down rather than up. Do you like this?

—Extend the code so you can control pitch separately from step duration.

—What can you do with a pair of sweeps that interact with each other?

—Add suitable graphics to the time bomb.

—Which obnoxious sounds are used how in ENGINE? How is flawless animation and thick sound achieved at the same time?

—How is the frog's voice produced in RIBBIT?

PROGRAM RM-4
OBNOXIOUS SOUNDS

```
----- NEXT OBJECT FILE NAME IS OBNOXIOUS SOUNDS
6800:            3              ORG  $6800     ; PUT MODULE #4 AT $6800


6800:            5 ;  ********************************************
6800:            6 ;  *                                          *
6800:            7 ;  *          -< OBNOXIOUS  SOUNDS >-          *
6800:            8 ;  *                                          *
6800:            9 ;  *          (CUSTOM CODING METHOD)           *
6800:           10 ;  *                                          *
6800:           11 ;  *      VERSION 1.0   ($6800-$687F)          *
6800:           12 ;  *                                          *
6800:           13 ;  *              11-24-82                     *
6800:           14 ;  *.........................................*
6800:           15 ;  *                                          *
6800:           16 ;  *          COPYRIGHT C 1982 BY             *
6800:           17 ;  *                                          *
6800:           18 ;  *     DON LANCASTER AND SYNERGETICS        *
6800:           19 ;  *     BOX 1300, THATCHER AZ., 85552        *
6800:           20 ;  *                                          *
6800:           21 ;  *     ALL COMMERCIAL RIGHTS RESERVED       *
6800:           22 ;  *                                          *
6800:           23 ;  ********************************************


6800:           25 ;         *** WHAT IT DOES ***

6800:           27 ; THIS MODULE GENERATES SIXTEEN DIFFERENT SOUND EFFECTS
6800:           28 ; FOR USE INSIDE ANOTHER PROGRAM.
6800:           29 ;
6800:           30 ;
6800:           31 ;
6800:           32 ;


6800:           34 ;         *** HOW TO USE IT ***

6800:           36 ; TO USE FROM MACHINE LANGUAGE, LOAD THE X REGISTER WITH
6800:           37 ; A SOUND SELECTION FROM $00 TO $1F AND THEN JSR TO $6824.
6800:           38 ;
6800:           39 ; TO USE FROM APPLESOFT, POKE 26659 WITH THE SOUND
6800:           40 ; EFFECT FROM 0-15 AND CALL 26658.
6800:           41 ;
```

PROGRAM RM-4, CONT'D . . .

```
6800:          44 ;              *** GOTCHAS ***

6800:          46 ;   THE X REGISTER IS DESTROYED BY THIS SUBROUTINE.
6800:          47 ;   REGISTERS P,Y, AND A ARE SAVED FOR YOU.
6800:          48 ;
6800:          49 ;   THE PROGRAM MUST BE PLACED IN A PROTECTED AREA
6800:          50 ;   IF IT IS TO BE USED BY EITHER BASIC.
6800:          51 ;


6800:          53 ;              *** ENHANCEMENTS ***

6800:          55 ;   YOU CAN CHANGE THE EFFECTS BY CHANGING THE TRIP AND
6800:          56 ;   SWEEP VALUES FOR EACH FILE SELECTION.  SEE THE
6800:          57 ;   EFFECT FILE LISTING FOR PRESENTLY AVAILABLE EFFECTS.
6800:          58 ;
6800:          59 ;   EXTRA TONES ARE EASILY ADDED BY LENGTHENING THE SOUND
6800:          60 ;   EFFECT FILES SEF0-SEF15 AND CHANGING FLNGTH4


6800:          62 ;              *** RANDOM COMMENTS ***

6800:          64 ;   TO ACTIVATE THE DEMO PROGRAM THAT PLAYS ALL SIXTEEN
6800:          65 ;   NOTES IN ORDER, USE JSR $6800 OR CALL 26624.
6800:          66 ;
6800:          67 ;
6800:          68 ;
6800:          69 ;
```

PROGRAM RM-4, CONT'D . . .

```
6800:              72 ;              *** HOOKS ***


FC58:              74 HOME    EQU  $FC58      ; CLEAR SCREEN
FB2F:              75 INIT    EQU  $FB2F      ; HOME CURSOR
C030:              76 SPKR    EQU  $C030      ; SPEAKER CLICK OUTPUT
FCA8:              77 WAIT    EQU  $FCA8      ; TIME DELAY SET BY ACCUMULATOR




6800:              79 ;              *** DEMO  ***
6800:              80 ;
6800:              81 ;


6800:              83 ;       THE DEMO PROGRAM PLAYS EACH OF THE SIXTEEN
6800:              84 ;       SOUND EFFECTS IN ORDER, SEPARATED BY A
6800:              85 ;       TIME DELAY.
6800:              86 ;
6800:              87 ;
6800:              88 ;


6800:20 2F FB      90 DEMO4   JSR  INIT       ; MAKE SCREEN BLANK
6803:20 58 FC      91         JSR  HOME       ;
6806:A9 00         92         LDA  #$00       ; START WITH FIRST NOTE
6808:48            93         PHA             ;  AND SAVE ON STACK

6809:AA            95 NXTNOT4 TAX             ;
680A:20 24 68      96         JSR  OBNOX4     ; AND PLAY IT
680D:A0 0A         97         LDY  #10        ; STALL FOR TIME

680F:20 A8 FC      99 STALL4  JSR  WAIT       ;
6812:88           100         DEY             ;
6813:D0 FA        101         BNE  STALL4     ; TILL DELAY DONE

6815:68           103         PLA             ; GET NOTE NUMBER
6816:CD 5F 68     104         CMP  FLNGTH4    ; DONE WITH LAST NOTE?
6819:F0 06        105         BEQ  DONE4      ;  YES, EXIT

681B:18           107         CLC             ;
681C:69 01        108         ADC  #$01       ;  NO, PICK NEXT NOTE
681E:48           109         PHA             ;
681F:D0 E8        110         BNE  NXTNOT4    ;  ALWAYS

6821:60           112 DONE4   RTS             ; AND EXIT
```

PROGRAM RM-4, CONT'D . . .

```
6822:              115 ;                 *** OBNOX MODULE ***
6822:              116 ;
6822:              117 ;


6822:              119 ;              THIS MODULE GENERATES THE SOUND EFFECTS IN
6822:              120 ;              EXCHANGE FOR AN X VALUE FROM $00 TO $0F.
6822:              121 ;
6822:              122 ;
6822:              123 ;
6822:              124 ;


6822:A2 00        126 BASENT4 LDX  #$00        ; BASIC POKE HERE+1
6824:08           127 OBNOX4  PHP              ; ML ENTRY POINT
6825:48           128         PHA              ;
6826:98           129         TYA              ; SAVE P,A, AND Y REGS
6827:48           130         PHA              ;

6828:8A           132         TXA              ; RANGE CHECK ON SELECTION
6829:CD 5F 68     133         CMP  FLNGTH4     ;   TO MAKE SURE ITS IN FILE
682C:90 02        134         BCC  LOK4        ;
682E:A9 00        135         LDA  #$00        ; DEFAULT TO ZERO SELECTION
6830:0A           136 LOK4    ASLA             ;   AND DOUBLE FILE POINTER
6831:AA           137         TAX              ;
6832:BD 60 68     138         LDA  SEF0,X      ; GET NUMBER OF TRIPS
6835:8D 5E 68     139         STA  TRPCNT4     ;   AND SAVE
6838:E8           140         INX              ;
6839:BD 60 68     141         LDA  SEF0,X      ; GET SWEEP RANGE
683C:8D 40 68     142         STA  SWEEP4+1    ;   AND SAVE

683F:A0 00        144 SWEEP4  LDY  #$00        ; SWEEP VALUE POKED HERE
6841:98           145 NXTSWP4 TYA              ;
6842:AA           146         TAX              ; DURATION
6843:98           147 NXTCYC4 TYA              ; PITCH
6844:20 A8 FC     148         JSR  WAIT        ;
6847:2C 30 C0     149         BIT  SPKR        ; WHAP SPEAKER
684A:E0 80        150         CPX  #$80        ; BYPASS IF GEIGER
684C:F0 0B        151         BEQ  EXIT4       ;   SPECIAL EFFECT
684E:CA           152         DEX              ;
684F:D0 F2        153         BNE  NXTCYC4     ; ANOTHER CYCLE
6851:88           154         DEY              ;
6852:D0 ED        155         BNE  NXTSWP4     ; GO UP IN PITCH
6854:CE 5E 68     156         DEC  TRPCNT4     ; MADE ALL TRIPS?

6857:D0 E6        158         BNE  SWEEP4      ;   NO, REPEAT

6859:68           160 EXIT4   PLA              ; RESTORE REGISTERS
685A:A8           161         TAY              ;
685B:68           162         PLA              ;
685C:28          163         PLP              ;
685D:60          164         RTS              ; AND EXIT
```

PROGRAM RM-4, CONT'D . . .

```
685E:              167 ;              *** STASH ***


685E:01            169 TRPCNT4 DFB  $01        ; TRIP COUNT DECREMENTED HERE
685F:10            170 FLNGTH4 DFB  $10        ; SIXTEEN AVAILABLE SOUNDS




6860:              172 ;              *** SOUND EFFECT FILES ***


6860:              174 ;     EACH NOTE TAKES A TRIP AND A SWEEP VALUE IN SEQUENCE.
6860:              175 ;
6860:              176 ;     ADD $80 TO NUMBER OF GEIGER CLICKS WANTED.
6860:              177 ;
6860:              178 ;
6860:              179 ;


6860:01 08        181 SEF0    DFB  $01,$08    ; TICK
6862:01 18        182 SEF1    DFB  $01,$18    ; WHOPIDOOP
6864:FF 01        183 SEF2    DFB  $FF,$01    ; PIP
6866:06 10        184 SEF3    DFB  $06,$10    ; PHASOR
6868:01 30        185 SEF4    DFB  $01,$30    ; MUSIC SCALE
686A:20 06        186 SEF5    DFB  $20,$06    ; SHORT BRASS
686C:70 06        187 SEF6    DFB  $70,$06    ; MEDIUM BRASS
686E:FF 06        188 SEF7    DFB  $FF,$06    ; LONG BRASS
6870:01 A0        189 SEF8    DFB  $01,$A0    ; GEIGER
6872:FF 02        190 SEF9    DFB  $FF,$02    ; GLEEP
6874:04 1C        191 SEF10   DFB  $04,$1C    ; GLISSADE
6876:01 10        192 SEF11   DFB  $01,$10    ; QWIP
6878:30 0B        193 SEF12   DFB  $30,$0B    ; OBOE
687A:30 07        194 SEF13   DFB  $30,$07    ; FRENCH HORN
687C:50 09        195 SEF14   DFB  $50,$09    ; ENGLISH HORN
687E:01 64        196 SEF15   DFB  $01,$64    ; TIME BOMB




*** SUCCESSFUL ASSEMBLY: NO ERRORS
```

MUSICAL SONGS

an upgrade of the original "red book tones" song and music maker

Come on, kiddies. If you are going to reinvent the wheel, please make the thing roughly circular and put an axle somewhere near the middle, preferably pointing in some more or less reasonable direction.

The wheel in this case is a music machine that easily and simply gives you an audio tone in exchange for pitch and duration values. There are so many utterly atrocious attempts at this that it is no longer even funny.

In particular . . .

> **A music making subroutine MUST have totally separate and totally isolated ways of entering pitch and duration.**
>
> **ANYTHING ELSE ISN'T EVEN WRONG!**

If the duration of your note changes when you change the pitch, your music maker is less than worthless. Flush it.

It turns out that a really great music making subroutine has existed since year one that uniquely solves the pitch and duration interaction

problem. The sub is called the *red book tones,* Woz wrote it, and it appears, of all places, in the original red book.

The red book tones are a "middleweight" technique that lets you create reasonable sounding monophonic music, as well as providing an easy way to pick up lots of different cue and prompt tones for other program uses. The original code, as it first appeared, was all of twenty-one bytes long!

Today, of course, you cannot write commercial software and get away with monophonic, fixed timbre, or constant volume sound effects. Use of multiple voices, variable volume, and duty-cycling is absolutely mandatory. But, just as LORES is an essential stepping stone to commercially useful graphics, the red book tones are a necessary learning experience along the way to top-notch musical effects.

While we will not be reinventing the wheel, we are going to add a hubcap, some chrome, and better bearings.

First, we all call the newer version REDTONE. As with the original, it gives you a constant frequency square-wave tone in exchange for pitch and duration values. We've put REDTONE into source code so you can relocate it anywhere you want. The original code sat on page zero and had some Applesloth compatibility problems. The obvious choice of page three is so overloaded these days, that it is best to have something you can put anywhere you want.

REDTONE saves all the working registers to avoid conflicts with your high level code. The pitch and duration values are also saved for you, so you needn't reload the same duration value over and over again for cues or prompts.

There is now a *silent* pitch value of $FF. This is most handy for rests and pauses. The silence is timed out to the same duration value any other note would be. As a convenience, the notes are echoed to the cassette output port. You can greatly improve the sound by going through a small hi-fi amplifier and larger speaker. Use standard audio cables.

The maximum duration on the original code was a little short, particularly when it came to playing whole notes at low tempos, so REDTONE has a feature called a *duration multiplier* that lets you extend the duration in binary mulitples. You now have all the duration range you could possibly ever use, plus a ridiculous bunch more.

And that just about covers the code improvements. We've also made two use improvements. The first involves better pitch accuracy, and the second lets your Assembler enter music in a sane and more or less musical way. For instance, a half note of middle C is entered as "C1,H,". There are no worries about funny numbers.

Tempo is presently set by changing a single value before assembly. You can easily upgrade to a "real time" tempo control. I've purposely left this as an "exercise for the student."

Pitch Accuracy

Most people try to set up a tone generator to make some certain pitch exactly hit some musical note. Then they go up and down the scale from there, trying to "fit" the notes to the 8-bit pitch values needed.

The problem is that this technique works well for some notes and poorly for others. Some notes just won't fit and will sound out of tune.

Some review. An *octave* is a 2:1 frequency change, and is just about as far as you can easily reach on a piano, say from middle C to the next higher C. People have messed with how many notes go where for a long time, but today, most everyone uses a compromise system called the *equally tempered* scale.

The equally tempered scale has twelve notes per octave. The notes in the "key" of C are called, C, C#, D, E, F, F#, G, G#, A, A#, B, and back again to the next C that's one octave higher. Note that there is no "E#" or "B#" as such. Other keys may name these notes differently and may start at a different point, but regardless of which key is in use, there are only twelve notes per octave.

The pitch of a note is related to that note's frequency, which is called out in hertz, or cycles per second. For instance, the pitch of the A above middle C is standardized to a frequency of 440 Hz.

Since the ear is a logarithmic type device, it expects low frequency differences between notes for the low notes, and high frequency differences between notes for the high notes. If you tried to create a linear "scale" that went, say 300, 350, 400, 450, 500, . . . etc. Hz, it would sound very weird indeed.

Unmusical, even.

To get a log spacing of 12 notes over one octave, each successive *equally tempered* note has to be the *twelfth root of two* higher in frequency. This is roughly a factor of 1.06. Each note ends up roughly 6 percent higher in frequency than its neighbor.

The interval from note to note is called a *semitone*. A semitone is the difference from one key to the immediate next one on a piano, regardless of key color. A semitone is also a 6 percent increase in frequency. A pitch change of one semitone is thus only a few hertz for low notes, but is very much more than this for high notes.

How accurate do the tones have to be? It turns out that very few people have what is called "absolute pitch," so if the whole song is uniformly mistuned too high or too low, nobody will be able to tell.

What counts is the relation between the notes, or "relative pitch," and here, things get sticky fast . . .

> **Few people can tell ABSOLUTE PITCH, so it really doesn't matter whether all the notes are exactly set to their intended absolute frequencies.**
>
> **Just about anybody can tell RELATIVE PITCH, so it is super important that the notes all sound good together.**

Thus, if an "A" is really 480 Hz rather than 440, the odds are high that nobody will notice on a stand-alone song. So long, of course, that all the *other* notes are equally offset from where they belong *by the same proportion*. What is critical is the *relative* frequency difference between "A" and "A#," or between any other notes.

How critical is critical? Musicians call one one-hundredth of a semitone a *cent*. A one cent frequency error is an error of just under 0.06 percent in the ratio of two notes. It turns out that the best musicians can just barely spot a one cent frequency error, while an average careful listener can spot a three cent error.

The trick is to get accurate relative notes consistent with a pitch word that is only 8 bits wide. If you just force any old note to be exact and then try to find magic values for the other notes, one or more of them will sound sour.

If you play with funny numbers long enough, you'll find that there is a little known but super important *series* of 8-bit pitch values that give far and away the most accurate notes you can possibly get using 8-bit values. Any other attempt at pitch values will fall short of this optimum series, and you'll get several sour notes.

Here's the magic series and the notes involved . . .

"MAGIC" 8-BIT PITCH VALUES	
232	(A)
219	(A#)
207	(B)
195	(C)
184	(C#)
174	(D)
164	(D#)
155	(E)
146	(F)
138	(F#)
130	(G)
123	(G#)
116	(A)

These notes are all accurate to better than three cents in relative pitch. Once again, this is a "magic" series. Any other choice of pitch values will give you at least one sour note. Note that the pitch values will set the time between speaker motions of REDTONE, so the *higher* the pitch value, the *lower* the pitch or frequency of the note you get. It takes two shoves, one forward and one backward, of the speaker cone, to generate one full cycle of a REDTONE square wave. The timbre you get is a "woody" one roughly akin to a clarinet or a stopped organ pipe.

The approximate notes you actually get with REDTONE are shown in parentheses. You can continue up in pitch, but you'll eventually pick up some sour notes on the way. Just divide each of the "magic" values by two for the next octave, and so on.

Note that you will only have seven or fewer bits of accuracy for these higher notes. Which means a few of them may be off in pitch. By the way, you also have an additional magic 8-bit pitch value of 246. This translates to a REDTONE "G#" or "Ab," and it seemed to make more sense to start at "A" instead.

Separating Pitch and Duration
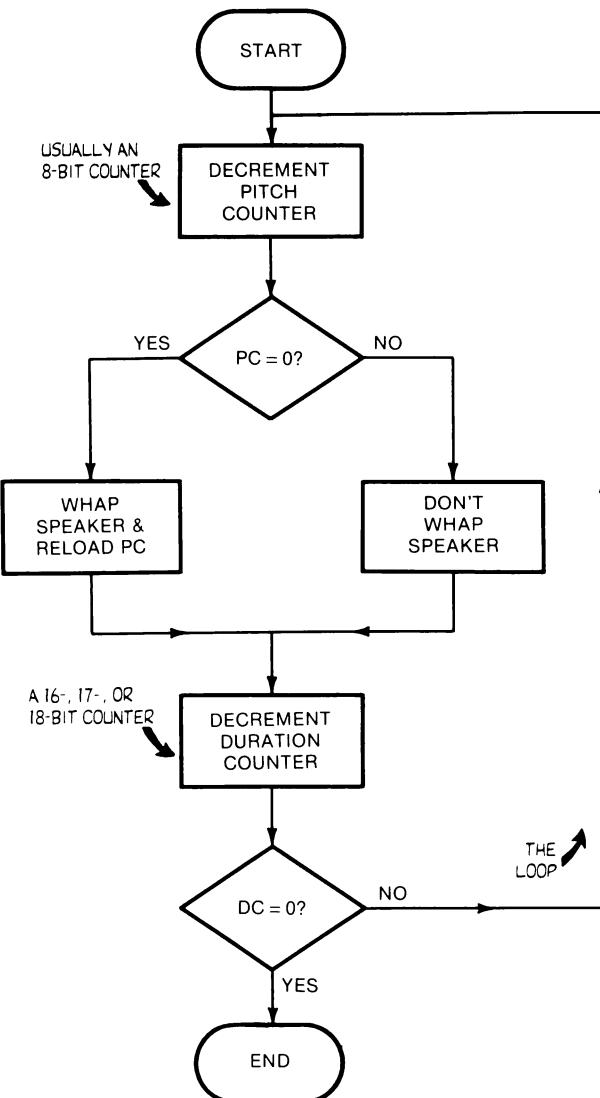
The "obvious" way to generate a tone is to count one register down to get the pitch. Each completed countdown whaps the speaker once. To get duration, you then count the total number of whappings.

Which is simple but wrong.

The trouble is that the high notes will sound much shorter than the low notes. Which gets to be a real mess. Any decent music maker subroutine *must* separate pitch and duration.

Here's how to do it . . .

USING A "SERVICE" LOOP TO SEPARATE PITCH & DURATION:

```
                          ┌─────────┐
                          │  START  │
                          └────┬────┘
                               │
  USUALLY AN          ┌────────┴────────┐
  8-BIT COUNTER ➘     │   DECREMENT     │
                      │     PITCH       │
                      │    COUNTER      │
                      └────────┬────────┘
                               │
              YES         ◇ PC = 0? ◇         NO
       ┌─────────────────┘         └─────────────────┐
       │                                             │
 ┌───────────┐                              ┌───────────┐
 │   WHAP    │                              │  DON'T    │
 │ SPEAKER & │                              │  WHAP     │
 │ RELOAD PC │                              │ SPEAKER   │
 └─────┬─────┘                              └─────┬─────┘
       └──────────────────┬──────────────────────┘
                          │
  A 16-, 17-, OR  ┌───────┴────────┐
  18-BIT COUNTER ➘│   DECREMENT    │
                  │    DURATION    │
                  │    COUNTER     │
                  └───────┬────────┘
                          │
                     ◇ DC = 0? ◇    NO ──────────► THE LOOP
                          │
                         YES
                          │
                     ┌─────────┐
                     │   END   │
                     └─────────┘
```

What you do is set up a tight *service loop* that *continuously* tests *both* the pitch and duration values. Two counters are involved, an 8-bit pitch counter, and a 16-bit or longer duration counter. The service loop continuously decrements *both* of these counters. When the magic pitch value is hit, the speaker gets whapped. When the magic duration value is hit, the tone ends. Since the duration values are usually much larger than the pitch values, you will normally get many pitch cycles in your note.

The way the original red book tones got its 16-bit duration values was to take an 8-bit duration value and multiply it by 256 using the Y register. Thus, the Y register had to go all the way around for each count of the duration counter.

All of which elegantly solved keeping pitch and duration separate.

A Duration Multiplier

The only little problem with this scheme was that 16 bits worth of duration weren't quite enough for some uses. Things were OK for simple songs, but for dotted half notes or for whole notes played at slow tempos, there simply wasn't enough duration to fully sound the note. The maximum duration was just under one second.

REDTONE gets around this by going to as many as 24 bits for the duration counter. It turns out that REDTONE never needs the accumulator, so this register is free to be used as a multiplying counter.

Here's how it works. You always initialize the accumulator to $00. Now, say you add some magic value to the accumulator and test for zero. The results you get depend on what you add. Four useful results include . . .

```
Adding $00 gives you

     00 00 00 00 00 00 00 00 00

                    and multiplies by ONE.

Adding $80 gives you

     00 80 00 80 00 80 00 80 00

                    and multiplies by TWO.

Adding $40 gives you

     00 40 80 C0 00 40 80 C0 00

                    and multiplies by FOUR.

Adding $20 gives you

     00 20 40 60 80 A0 C0 E0 00

                    and multiplies by EIGHT.
```
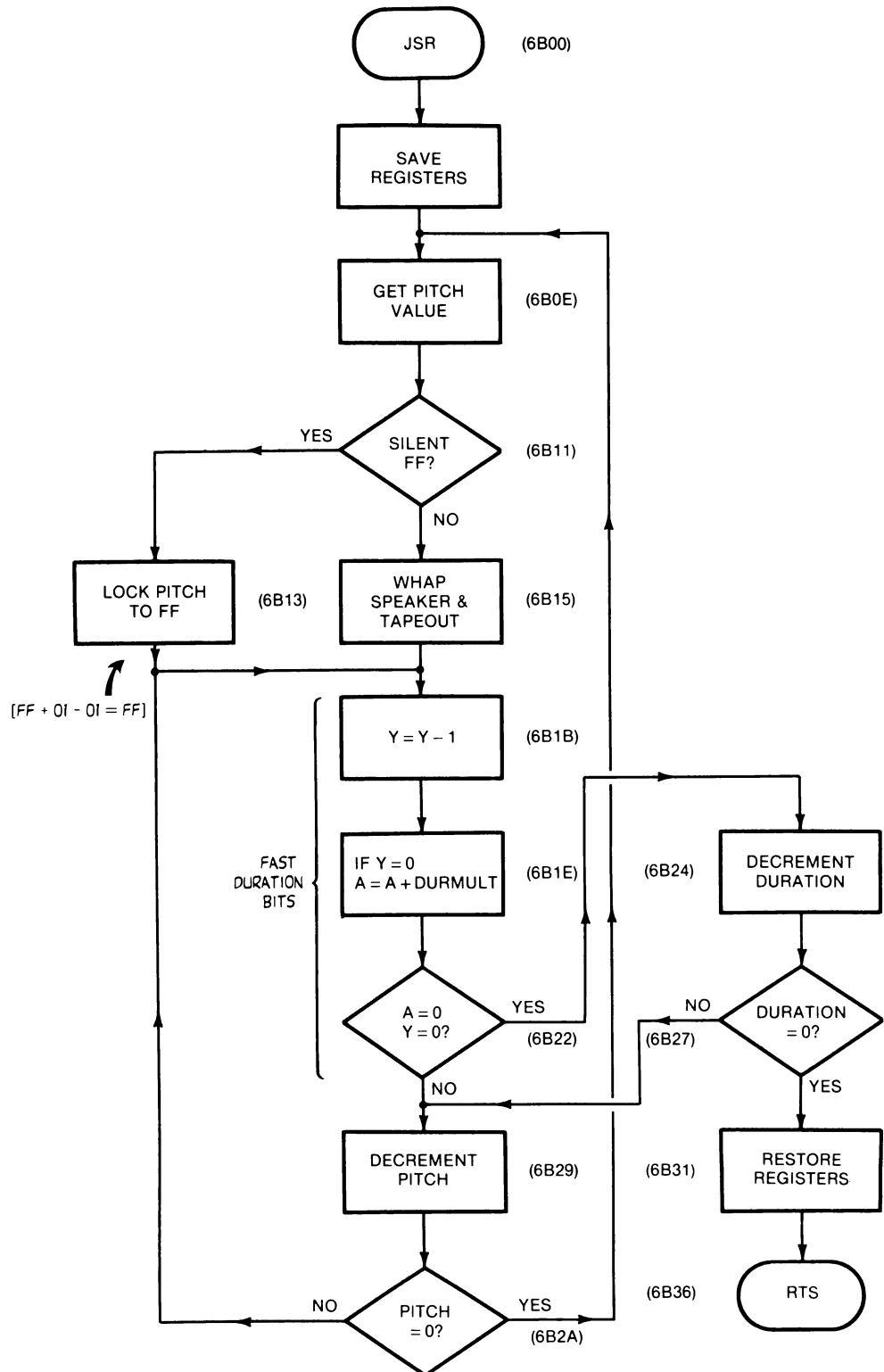
What you do is count down the duration counter every time you get a zero result. Thus, the $40 adder only decrements the duration counter on every *fourth* trip through the service loop. This makes the note last four times longer.

Usually, a "X2" multiplier is just what you need for most music. You can go up to "X256" multiplication, using an $01 magic value, if you want to get ridiculous.

With these details out of the way, let's look at the REDTONE subroutine. Here's the flowchart . . .

REDTONE FLOWCHART:

```
                    ( JSR )          (6B00)
                       |
                 [SAVE REGISTERS]
                       |
          +----------->[GET PITCH VALUE]   (6B0E)
          |            |
          |         < SILENT FF? >  (6B11)
          |      YES /         \ NO
          |         /           \
   [LOCK PITCH   (6B13)      [WHAP SPEAKER   (6B15)
    TO FF]                    & TAPEOUT]
   [FF+01-01=FF]                 |
          |                  [Y = Y-1]   (6B1B)
          |                      |
          |              [IF Y=0 A=A+DURMULT]  (6B1E)
          |                      |
          |                  < A=0 Y=0? >  (6B22)
          |              YES /       \ NO
          |                         [DECREMENT PITCH]  (6B29)
          |                            |
          +------ NO < PITCH=0? > YES  (6B2A)
```

FAST DURATION BITS

DECREMENT DURATION (6B24)
< DURATION =0? > (6B27)
RESTORE REGISTERS (6B31)
RTS (6B36)

You enter REDTONE with a pitch value of PITCH5 and a duration value of DURAT5. These values are not destroyed should you want to reuse them for simple prompts. You must also have a multiplier value

in DURMULT5, but leaving this value at $80 will give good results for most uses.

The registers are first saved. Then DURAT5 is copied into DURCNT5 where it can be counted down. The copying saves you having to reenter the same duration each time for simple prompts. This is followed by clearing the Y register and the accumulator. The accumulator will be used for the duration 1-2-4 multiplier, while the Y register will be used to scale the duration by 256. You can alternately use the Y register to adjust tempo in real time.

The pitch value is placed in the X register and tested. If the pitch is not $FF, the note is accepted and processed as usual. The speaker is then whapped, and then is echoed to the cassette output.

Next, the service loop takes over. First, the Y register is decremented. If Y hits zero, then the duration multiplier gets activated, by adding the multiplier value and testing for a zero result. If the Y register has gone all the way around and if the duration multiplier gives you a zero result, then, and only then, is DURCNT5 decremented. Note that this has the effect of multiplying the DURCNT5 value first by 256 and then by the accumulator multiplier of 1, 2, 4, or whatever.

If we have not gotten a zero duration value, we then knock one off the pitch counter and repeat the service loop process. The speaker gets whapped only on zero values of the pitch counter. Thus a single service loop separately keeps track of pitch and duration with only negligible interaction.

Note that the duration is the product of three 8-bit values. Duration is set by multiplying the Y register *times* DURAT5 *times* DURMULT5.

Every pitch zero, the speaker gets hit, and the X register gets reloaded with a new pitch value. The only exit from all this happens when DURCNT5 finally hits zero. At that point, the registers are restored and the subroutine exits to your calling code.

One final detail. If your chosen pitch value is $FF, the speaker is not sounded. This gives you a silent note, a pause, or a rest.

The side loop at LOCKX handles this detail for you. IF the pitch is $FF, the pitch is incremented to $00 by LOCKX, and then later decremented back to $FF in the main service loop. Thus a $FF pitch value *stays* at $FF all the way through the duration timing. This happens because $FF + $01 − $01 = $FF. A sounding pitch value gets counted down to zero and whaps the speaker every trip. A silent pitch value stays at $FF and bypasses the speaker, producing no sound.

A Demo or Two

The SONGPLY demo exercises REDTONE for you, playing that ever favorite song that Tarzan used to sing during his zebra maintenance days. SONGPLY works by picking pitch and duration value out of a songfile called TARZAN.

We have used a 16-bit full wide pointer to access the song file, so you can have more than 128 notes total in your song. This pointer is called NOTEP and is page zero stashed at $EF and $F0. To link SONGPLY to different songs, you change these pointers as needed.

The pause generator inside SONGPLY gives you a brief pause between notes that is set by PAUSE. Experiment to get the best results. The minimum PAUSE value is $01. Do not use $00!

Should your particular song demand some notes that slur or tie together, just use a minimum value of PAUSE, say $01. Then use sixteenths rests or whatever between those notes that do not slur.

Be sure to study the source code on SONGPLY very carefully, for it shows you a fairly friendly way to use labels to simplify writing your own songs. We'll leave details on this for you to puzzle out.

One tip. Use two $00 for END values at the end of your note file. That way, should you have an error in your list, you will still stop, catching the second END value.

Where to from here? We have thrown in a quick tester called the TIMBRE TESTER that will get you started in experimenting with different "voices" for your Apple. To use the TIMBRE TESTER, just put a number series into TIMBFLE and the number of numbers into TFLEN. You can get the magic numbers by trial and error, from a venture into Fourier Series (gulp!), or from full-fledged duty cycling experiments.

TIMBRE TESTER works by generating a waveform with many possible zero crossings. As you change the number of zero crossings and the spacing between them, the harmonic content of the note changes, giving you different "voicing" for your Apple.

As examples, a waveform that has very strong fourth, fifth, and sixth harmonics, with a very weak fundamental, second, and third, will sound as a three note major triad chord. Other pleasant two note effects include a strong second and third, third and fourth, fourth and fifth, second and fifth, and third and fifth harmonics. A note with no low harmonics except for the fundamental will voice as a pure and flute-like sinewave.

You can get these harmonics the way you want them, either by trial and error, or else by fancy math.

With duty cycling, you use lots of very high frequency cycles, set up so that the *average* speaker cone position matches the waveform you are trying to generate. You can easily get pure sinewaves, variable volume, and even exceptionally good human voice synthesis with fancy enough duty cycling.

Each value in TIMBFLE specs the delay time in microseconds, multiplied by five, between cone whappings. For a "fat" sound, you whap the cone many times per frequency cycle. As a fine point, knock two off each VOICE5 value, except for the last one. Knock four off it. Why?

The TIMBRE TESTER can easily give you string and woodwind-style tones, flutelike sinewaves, bells, two notes at once, three notes at once, "noisy" sounds, and even voice. All it takes is the right numbers in the right order.

Finding them is half the fun.

MIND BENDERS

—Extend TARZAN by entering the "hard parts" that I left out.

—Write your own songs for SONGPLY.

—Modify SONGPLY so you can control the tempo from a game paddle. Hint: Put the tempo into the Y register.

—Examine the *exact* timing involved in REDTONE. What effects do slight variations from "perfect" loop timing have?

—Show why LOCKX is not needed and how to replace it.

—Play "Applesoft" by using $D000 as NOTEFL. Why are the results no longer equally tempered?

—Modify REDTONE for a string voice with an 8:1 duty cycle.

—Show a two byte change to SONGPLY that lets you edit by playing one note at a time.

—Use the TIMBRE TESTER to produce a pure sinewave, and then two notes at once. Then, ring a bell.

—Some poor attempts at duty cycling may buzz or whine. Why? How can you eliminate this?

PROGRAM RM-5
MUSICAL SONGS

```
----- NEXT OBJECT FILE NAME IS MUSICAL SONGS
6900:            3              ORG  $6900      ; PUT MODULE #5 AT $6900


6900:            5 ; ********************************************
6900:            6 ; *                                          *
6900:            7 ; *            -< MUSICAL SONGS >-            *
6900:            8 ; *                                          *
6900:            9 ; *         ( MODIFIED RED BOOK TONES )       *
6900:           10 ; *                                          *
6900:           11 ; *       VERSION 1.0   ($6900-$6B3A)        *
6900:           12 ; *                                          *
6900:           13 ; *                5-24-83                   *
6900:           14 ; *...................................*
6900:           15 ; *                                          *
6900:           16 ; *           COPYRIGHT C 1983 BY            *
6900:           17 ; *                                          *
6900:           18 ; *      DON LANCASTER AND SYNERGETICS       *
6900:           19 ; *      BOX 1300, THATCHER AZ., 85552       *
6900:           20 ; *                                          *
6900:           21 ; *      ALL COMMERCIAL RIGHTS RESERVED      *
6900:           22 ; *                                          *
6900:           23 ; ********************************************


6900:           25 ;            *** WHAT IT DOES ***

6900:           27 ; THIS MODULE SHOWS YOU HOW TO USE THE MODIFIED
6900:           28 ; RED BOOK TONE SUBROUTINE TO PLAY MUSICAL SONGS.
6900:           29 ;
6900:           30 ; THERE IS ALSO A TIMBER TESTER FOR EVALUATION
6900:           31 ; OF SPECIAL APPLE VOICES AND SOUND EFFECTS.


6900:           33 ;            *** HOW TO USE IT ***

6900:           35 ; TO PLAY A SINGLE NOTE:
6900:           36 ;     PUT YOUR PITCH IN PITCH5 AT $6B3A (27450).
6900:           37 ;     PUT THE DURATION IN DURAT5 AT $6B37 (27447).
6900:           38 ;     THEN JSR REDTONE AT $6B00 (27392).


6900:           40 ; TO PLAY YOUR OWN SONG:
6900:           41 ;     PUT THE STARTING ADDRESS OF YOUR SONG INTO
6900:           42 ;     SONGLOC AND SONGLOC+1 AT $6944 AND $6945.
6900:           43 ;     THEN JSR SONGPLY AT $6910.  APPLESLOTH
6900:           44 ;     EQUIVALENTS ARE 26948, 26949, AND 26896.


6900:           46 ; TO PLAY TARZAN:
6900:           47 ;     DO A JSR $6900 OR CALL 26880.
```

PROGRAM RM-5, CONT'D . . .

```
6900:          50 ;              *** GOTCHAS ***

6900:          52 ;   A CHANGE OF TEMPO PRESENTLY NEEDS REASSEMBLY.
6900:          53 ;   REDTONE IS LIMITED TO "WOODWIND" SQUARE WAVES.


6900:          55 ;              *** ENHANCEMENTS ***

6900:          57 ;   THIS SOURCE CODE ALSO SHOWS YOU HOW TO COMPOSE
6900:          58 ;   YOUR OWN SONGS IN THE EQUALLY TEMPERED MUSICAL
6900:          59 ;   SCALE BY USING LABELS THAT SIMPLIFY NOTE ENTRY.
6900:          60 ;


6900:          62 ;              *** RANDOM COMMENTS ***

6900:          64 ;   THIS IS "MIDDLEWEIGHT" CODE INTENDED TO SHOW
6900:          65 ;   PROGRAMMING SKILLS AND TECHNIQUES.  FANCIER
6900:          66 ;   METHODS SHOULD BE USED FOR COMMERCIAL PROGRAMS
6900:          67 ;   OR FOR SERIOUS MUSICAL COMPOSITION.
6900:          68 ;


6900:          70 ;              *** HOOKS ***

FC58:          72 HOME     EQU   $FC58     ; CLEAR TEXT SCREEN AND HOME CURSOR
FB2F:          73 INIT     EQU   $FB2F     ; INITIALIZE TEXT SCREEN
C010:          74 KBDSTR   EQU   $C010     ; KEYBOARD STROBE
C000:          75 IOADR    EQU   $C000     ; KEYBOARD INPUT LOCATION
C030:          76 SPKR     EQU   $C030     ; SPEAKER CLICK OUTPUT
C020:          77 TAPEOUT  EQU   $C020     ; CASSETTE TAPE OUT (TONE ECHO)
FCA8:          78 WAIT     EQU   $FCA8     ; MONITOR TIME DELAY


00EF:          80 NOTEP    EQU   $EF       ; NOTE POINTER PAIR FOR SONGLPLY
```

PROGRAM RM-5, CONT'D . . .

```
69C0:          83 ;                    *** CONSTANTS ***


6900:          85 ;         PITCH LABELS USED FOR SONG COMPOSITION
6900:          86 ;
6900:          87 ;         FOR BEST SOUND, ALWAYS USE THE NOTE
6900:          88 ;         VALUES NEAREST THE TOP OF THIS LIST.



00E8:          90 A1    EQU  232        ; NOTE A BELOW MIDDLE C

00DB:          92 A1S   EQU  219        ;      A#
00DB:          93 B1F   EQU  219        ;      Bb
00CF:          94 B1    EQU  207        ;      B
00C3:          95 C1    EQU  195        ;      C
00B8:          96 C1S   EQU  184        ;      C#
00B8:          97 D1F   EQU  184        ;      Db
00AE:          98 D1    EQU  174        ;      D
00A4:          99 D1S   EQU  164        ;      D#
00A4:         100 E1F   EQU  164        ;      Eb
009B:         101 E1    EQU  155        ;      E
0092:         102 F1    EQU  146        ;      F
008A:         103 F1S   EQU  138        ;      F#
008A:         104 G1F   EQU  138        ;      Gb
0082:         105 G1    EQU  130        ;      G
007B:         106 G1S   EQU  123        ;      G#
007B:         107 A1F   EQU  123        ;      Ab

0074:         109 A2    EQU  116        ; NOTE A ABOVE MIDDLE C

006E:         111 A2S   EQU  110        ;      A#
006E:         112 B2F   EQU  110        ;      Bb
0067:         113 B2    EQU  103        ;      B
0062:         114 C2    EQU  98         ;      C
005C:         115 C2S   EQU  92         ;      C#
005C:         116 D2F   EQU  92         ;      Db
0057:         117 D2    EQU  87         ;      D
0052:         118 D2S   EQU  82         ;      D#
0052:         119 E2F   EQU  82         ;      Eb
004E:         120 E2    EQU  78         ;      E
0049:         121 F2    EQU  73         ;      F
0045:         122 F2S   EQU  69         ;      F#
0045:         123 G2F   EQU  69         ;      Gb
0041:         124 G2    EQU  65         ;      G
003D:         125 G2S   EQU  61         ;      G#
003D:         126 A2F   EQU  61         ;      Ab

003A:         128 A3    EQU  58         ; SECOND A ABOVE MIDDLE C


00FF:         130 R     EQU  $FF        ; SILENT OR REST
0000:         131 END   EQU  $00        ; END OF SONG   (USE TWICE!)
```

PROGRAM RM-5, CONT'D . . .

```
6900:          134 ;          DURATION LABELS USED FOR SONG COMPOSITION


6900:          136 ;            A REPEAT ASSEMBLY PASS IS NEEDED AT
6900:          137 ;            PRESENT FOR EACH CHANGE IN TEMPO.



0009:          139 TEMPO  EQU  $09      ; MASTER TEMPO CONTROL   ($0F MAXIMUM!)
0080:          140 MULT   EQU  $80      ; TEMPO MULTIPLIER (00=X1 $80=X2 $40=X4
0048:          141 PAUSE  EQU  $48      ; INTERNOTE PAUSE TIME


0009:          143 S      EQU  TEMPO*1  ; SIXTEENTH NOTE
000D:          144 DS     EQU  S/2+S    ; DOTTED SIXTEENTH
0012:          145 E      EQU  TEMPO*2  ; EIGHTH NOTE
001B:          146 DE     EQU  TEMPO*3  ; DOTTED EIGHTH
0024:          147 Q      EQU  TEMPO*4  ; QUARTER NOTE
0036:          148 DQ     EQU  TEMPO*6  ; DOTTED QUARTER
0048:          149 H      EQU  TEMPO*8  ; HALF NOTE
006C:          150 DH     EQU  TEMPO*12 ; DOTTED HALF NOTE
0090:          151 W      EQU  TEMPO*16 ; WHOLE NOTE
```

PROGRAM RM-5, CONT'D . . .

```
6900:              155 ;        ***  MUSICAL SONGS  ***
6900:              156 ;
6900:              157 ;   THIS SUBROUTINE USES REDTONE TO PLAY THE
6900:              158 ;   SONG WHOSE STARTING ADDRESS IS IN SONGLOC.


6900:20 2F FB     160 TAR      JSR  INIT       ; INITIALIZE TEXT SCREEN
6903:20 58 FC     161          JSR  HOME       ;  AND CLEAR IT

6906:A9 46        163          LDA  #>TARZAN   ; TO PLAY TARZAN ONLY
6908:8D 44 69     164          STA  SONGLOC    ;
690B:A9 69        165          LDA  #<TARZAN   ;
690D:8D 45 69     166          STA  SONGLOC+1  ;


6910:AD 44 69     168 SONGPLY  LDA  SONGLOC    ; MOVE SONG ADDRESS TO POINTER
6913:85 EF        169          STA  NOTEP      ;
6915:AD 45 69     170          LDA  SONGLOC+1  ; POSITION THEN PAGE AS USUAL
6918:85 F0        171          STA  NOTEP+1    ;

691A:A0 00        173          LDY  #$00       ; FOR PURE INDIRECT
691C:A9 80        174          LDA  #MULT      ; SET DURATION MULTIPLIER
691E:8D 39 6B     175          STA  DURMUL5    ; AND POKE TO REDTONE

6921:B1 EF        177 MORE5    LDA  (NOTEP),Y  ; GET PITCH VALUE
6923:F0 1E        178          BEQ  DONE5      ; EXIT IF END
6925:8D 3A 6B     179          STA  PITCH5     ; POKE PITCH
6928:E6 EF        180          INC  NOTEP      ; GO TO NEXT FILE VALUE
692A:D0 02        181          BNE  NOCY5      ;  PAGE OVERFLOW?
692C:E6 F0        182          INC  NOTEP+1    ;   YES
692E:B1 EF        183 NOCY5    LDA  (NOTEP),Y  ; GET DURATION VALUE
6930:8D 37 6B     184          STA  DURAT5     ; STASH DURATION VALUE
6933:20 00 6B     185          JSR  REDTONE    ; PLAY THE NOTE
6936:A9 48        186          LDA  #PAUSE     ; GET INTERNOTE DELAY
6938:20 A8 FC     187          JSR  WAIT       ; AND DELAY
693B:E6 EF        188          INC  NOTEP      ; GO TO NEXT FILE VALUE
693D:D0 E2        189          BNE  MORE5      ;  PAGE OVERFLOW?
693F:E6 F0        190          INC  NOTEP+1    ;   YES
6941:D0 DE        191          BNE  MORE5      ; ALWAYS (WELL, ALMOST!)

6943:60           193 DONE5    RTS             ; END OF SONG
```

PROGRAM RM-5, CONT'D . . .

```
6944:            196 ;           *** SONG POINTER STASH ***

6944:46 69      198 SONGLOC DFB  >TARZAN,<TARZAN


6946:            200 ;           *** SONG FILE ***


6946:            202 ; EACH NOTE IS ENTERED, PITCH FIRST AND
6946:            203 ; DURATION SECOND USING LABELS AS SHOWN.
6946:            204 ;


6946:74 48 74   206 TARZAN  DFB          A2,H,A2,H,G1,Q,F1S,Q,F1S,H
6949:48 82 24
694C:8A 24 8A
694F:48
6950:92 24 8A   207         DFB          F1,Q,F1S,Q,F1S,W,R,H
6953:24 8A 90
6956:FF 48
6958:92 24 8A   208         DFB          F1,Q,F1S,Q,F1S,H,F1,Q,F1S,Q
695B:24 8A 48
695E:92 24 8A
6961:24
6962:74 48 8A   209         DFB          A2,H,F1S,DQ,A2,E,G1,W,E1,DH,R,Q
6965:36 74 12
6968:82 90 9B
696B:6C FF 24
696E:9B 48 A4   210         DFB          E1,H,D1S,Q,E1,Q,E1,H
6971:24 9B 24
6974:9B 48
6976:A4 24 9B   211         DFB          D1S,Q,E1,Q,A2,W,R,H
6979:24 74 90
697C:FF 48
697E:8A 24 9B   212         DFB          F1S,Q,E1,Q,F1S,Q,A2,DH
6981:24 8A 24
6984:74 6C
6986:67 6C 67   213         DFB          B2,DH,B2,Q,E1,W,R,H
6989:24 9B 90
698C:FF 48
698E:74 48 74   214         DFB          A2,H,A2,H,G1,Q,F1S,Q,F1S,H
6991:48 82 24
6994:8A 24 8A
6997:48
6998:92 24 8A   215         DFB          F1,Q,F1S,Q,F1S,W,R,H
699B:24 8A 90
699E:FF 48
```

PROGRAM RM-5, CONT'D . . .

```
69A0:92 24 8A    218         DFB         F1,Q,F1S,Q,F1S,H,F1,Q,G1,Q
69A3:24 8A 48
69A6:92 24 82
69A9:24
69AA:82 24 8A    219         DFB         G1,Q,F1S,Q,E1,DQ,C2S,E,E1,DH,D1,H
69AD:24 9B 36
69B0:5C 12 9B
69B3:6C AE 48
69B6:FF 24 AE    220         DFB         R,Q,D1,Q,D1,H,C1S,Q,D1,Q
69B9:24 AE 48
69BC:B8 24 AE
69BF:24
69C0:92 48 9B    221         DFB         F1,H,E1,Q,D1,Q,D2,W,R,Q
69C3:24 AE 24
69C6:57 90 FF
69C9:24
69CA:92 24 9B    222         DFB         F1,Q,E1,Q,F1S,Q,A2,E,R,E,D1,Q
69CD:24 8A 24
69D0:74 12 FF
69D3:12 AE 24
69D6:9B 24 8A    223         DFB         E1,Q,F1S,Q,A2,E,R,E
69D9:24 74 12
69DC:FF 12
69DE:E8 24 CF    224         DFB         A1,Q,B1,Q,F1S,Q,E1,W,D1,Q
69E1:24 8A 24
69E4:9B 90 AE
69E7:24
69E8:00 00       225         DFB         END,END
```

PROGRAM RM-5, CONT'D . . .

```
69EA:          228 ;                    *** TIMBRE TESTER ***
69EA:          229 ;
69EA:          230 ;      THIS ROUTINE LETS YOU EVALUATE SPECIAL VOICES,
69EA:          231 ;      DUTY CYCLING, MULTI-TONES AND OTHER EFFECTS.
69EA:          232 ;
69EA:          233 ;      TO USE, LOAD TIMBFLE WITH THE DELAY VALUES
69EA:          234 ;      BETWEEN ZERO CROSSINGS.  LOAD TFLENGTH WITH
69EA:          235 ;      THE NUMBER OF ZERO CROSSINGS PER FUNDAMENTAL
69EA:          236 ;      NOTE CYCLE.
69EA:          237 ;
69EA:          238 ;      TO RUN:
69EA:          239 ;
69EA:          240 ;         JSR $6AC0 FROM MACHINE LANGUAGE
69EA:          241 ;         CALL 27328 FROM APPLESLOTH.
69EA:          242 ;
69EA:          243 ;         EXIT ON ANY KEY PRESSED.


6AC0:          245           ORG  TAR+$01C0 ; LEAVE ROOM FOR SONG FILES

6AC0:2C 10 C0  247 TIMBRE  BIT  KBDSTR    ; RESET KEYBOARD
6AC3:AE DE 6A  248 RESCAN5 LDX  TFLENGTH  ; START NEW SCAN
6AC6:CA        249 NEXT5   DEX            ; NEXT VALUE
6AC7:30 FA     250         BMI  RESCAN5   ; RESET IF COMPLETE
6AC9:BC DF 6A  251         LDY  TIMBFLE,X ; GET DELAY VALUE
6ACC:88        252 LOOP5   DEY            ; DELAY 5N+1 CYCLES
6ACD:D0 FD     253         BNE  LOOP5     ; STALL FOR TIME
6ACF:2C 30 C0  254         BIT  SPKR      ; WHAP SPEAKER
6AD2:2C 20 C0  255         BIT  TAPEOUT   ; WHAP CASSETTE OUTPUT
6AD5:2C 00 C0  256         BIT  IOADR     ; KEYPRESSED?
6AD8:10 EC     257         BPL  NEXT5     ; REPEAT IF NO KP
6ADA:2C 10 C0  258         BIT  KBDSTR    ; RESET KEYSTROBE
6ADD:60        259         RTS            ; AND EXIT


6ADE:          261 ;      *** TIMBRE DELAY VALUES ***


6ADE:08        263 TFLENGTH DFB $08       ; NUMBER OF CROSSINGS IN TIMBFLE


6ADF:60 6A 6A  265 TIMBFLE DFB            $60,$6A,$6A,$27
6AE2:27
6AE3:27 6A 6A  266         DFB            $27,$6A,$6A,$5E
6AE6:5E
```

PROGRAM RM-5, CONT'D . . .

```
6AE7:              269 ;         *** MODIFIED RED BOOK TONE SUBROUTINE ***


6AE7:              271 ;         A JSR TO REDTONE PLAYS A SINGLE NOTE.
6AE7:              272 ;
6AE7:              273 ;         THE PITCH MUST BE PREPLACED IN PITCH5
6AE7:              274 ;         DURATION MUST BE PREPLACED IN DURAT5.
6AE7:              275 ;
6AE7:              276 ;         A PITCH5 VALUE OF $FF IS SILENT.


6B00:              278             ORG  TAR+$0200 ; LEAVE ROOM FOR TIMBRE FILES

6B00:48            280 REDTONE PHA             ; SAVE REGISTERS
6B01:98            281         TYA             ;
6B02:48            282         PHA             ;
6B03:8A            283         TXA             ;
6B04:48            284         PHA             ;
6B05:AD 37 6B      285         LDA  DURAT5      ; MOVE DURATION VALUE TO
6B08:8D 38 6B      286         STA  DURCNT5     ;  COUNTABLE LOCATION

6B0B:A0 00         288         LDY  #$00        ; INIT FAST DURATION COUNTER
6B0D:98            289         TYA              ; INIT DURATION MULTIPLIER

6B0E:AE 3A 6B      291 WHAP    LDX  PITCH5      ; GET PITCH VALUE
6B11:E0 FF         292         CPX  #$FF        ; IS IT SILENT?
6B13:F0 19         293         BEQ  LOCKX       ; YES, KEEP IT SILENT
6B15:2C 30 C0      294         BIT  SPKR        ; WHAP SPEAKER
6B18:2C 20 C0      295         BIT  TAPEOUT     ; AND ECHO TO CASSETTE OUTPUT
6B1B:88            296 NOWHAP  DEY              ; DECREMENT FAST DURATION COUNT
6B1C:D0 0B         297         BNE  NOC5        ; IF NO BORROW
6B1E:18            298         CLC              ;
6B1F:6D 39 6B      299         ADC  DURMUL5     ; DURATION MULTIPLIER
6B22:D0 05         300         BNE  NOC5        ;   IGNORE ALL BUT ZERO RESULTS
6B24:CE 38 6B      301         DEC  DURCNT5     ; DECREMENT SLOW DURATION
6B27:F0 08         302         BEQ  EXIT5       ; IF FINISHED
6B29:CA            303 NOC5    DEX              ; DECREMENT PITCH VALUE
6B2A:D0 EF         304         BNE  NOWHAP      ; PITCH NOT DONE
6B2C:F0 E0         305         BEQ  WHAP        ; PITCH DONE, ALWAYS TAKEN

6B2E:E8            307 LOCKX   INX              ; TRAP X TO $FF
6B2F:F0 EA         308         BEQ  NOWHAP      ; ALWAYS TAKEN

6B31:68            310 EXIT5   PLA              ; RESTORE REGISTERS
6B32:AA            311         TAX              ;
6B33:68            312         PLA              ;
6B34:A8            313         TAY              ;
6B35:68            314         PLA              ;
6B36:60            315         RTS              ; AND EXIT
```

PROGRAM RM-5, CONT'D . . .

```
6B37:            318 ;              *** REDTONE STASH ***

6B37:72          320 DURAT5  DFB  $72      ; DURATION GOES HERE
6B38:72          321 DURCNT5 DFB  $72      ;  GETS COUNTED HERE
6B39:80          322 DURMUL5 DFB  $80      ; DURATION MULTIPLIER
6B3A:72          323 PITCH5  DFB  $72      ; PITCH GOES HERE
```

*** SUCCESSFUL ASSEMBLY: NO ERRORS

6

OPTION PICKER

a general and flexible way to handle menu selections and in-program jumps

Just about any larger program eventually gets to a point where it has to jump six ways from Sunday. These may involve internal jumps, such as when an adventure decides it has to check to be sure the giant armadillo is awake. Or, they might involve user input, such as a menu selection, the "T" for trace command of a monitor, or the "[S]" save command of a word processor.

A code module that lets a program go to one of many possible tasks is called an *option picker . . .*

> **OPTION PICKER—**
>
> **A code module that lets a program continue by jumping to a selected one of many possible tasks.**

Now, option picking doesn't sound like a very big deal. The trick is to come up with one single option picker that you can adapt to any program you want to, while keeping things as short and as flexible as possible.

Here is the obvious way to pick one of a few different options . . .

```
OPTION   JSR   GETKEY   ; GET USER SELECTION

TRYA     CMP   $C1      ; AN A?
         BNE   TRYB     ;   NO, KEEP TRYING
         JMP   TASKA    ;   YES, GOTOA

TRYB     CMP   $C2      ; A B?
         BNE   TRYC     ;   NO, KEEP TRYING
         JMP   TASKB    ;   YES, GOTOB

TRYC     CMP   $C3      ; A C?
         BNE   MISSED   ;   NO, WE LOST
         JMP   TASKC    ;   YES, GOTOC

MISSED   JMP   ERRPROC  ; NO VALID SELECTION, TRY AGAIN
```

What you do here is get a user input and then check for a valid key-pressed entry. In this example, we test for a capital A, B, or C. But you can test for any character in any order, such as for Y, N, or [ESC]. The testing is done by comparing the hit key against the ASCII codes for each key.

Now, this is a useful option picker, and, in fact, may be the best one if you have eight or fewer unordered key selections. But this simple option picker has several grevious faults.

The most obvious fault is that the code gets ridiculously long for lots of different selections. One poor "solution" to this is to force sequential responses, such as 0-9 or A-G, and then calculate your response by subtracting the ASCII value of the selection from the ASCII value of the lowest possible selection. This then gives you a straight binary value that you use as an index to grab an address.

But, forcing users to relate to sequential numbers is a bad scene. Which gets worse when you just have to have a tenth or eleventh selection and no way to get it with a single keystroke. Insisting that users relate to sequential letters is even worse. It is just plain poor design to require a "B" selection for a "SAVE" command, and so on. Using either "S" or "[S]" for a save is vastly better.

Another problem of requiring only sequential selections is that there is no way to pick up one or two oddball control characters responses, such as an [ESC] or a [C].

Thus . . .

> **AVOID forcing users to pick from 0-9 or between A-H as input selections.**
>
> **Instead, try to make each selection meaningful to the user in some way.**

So, forced 0-9 or A-H responses have their problems and should be avoided for use in option pickers.
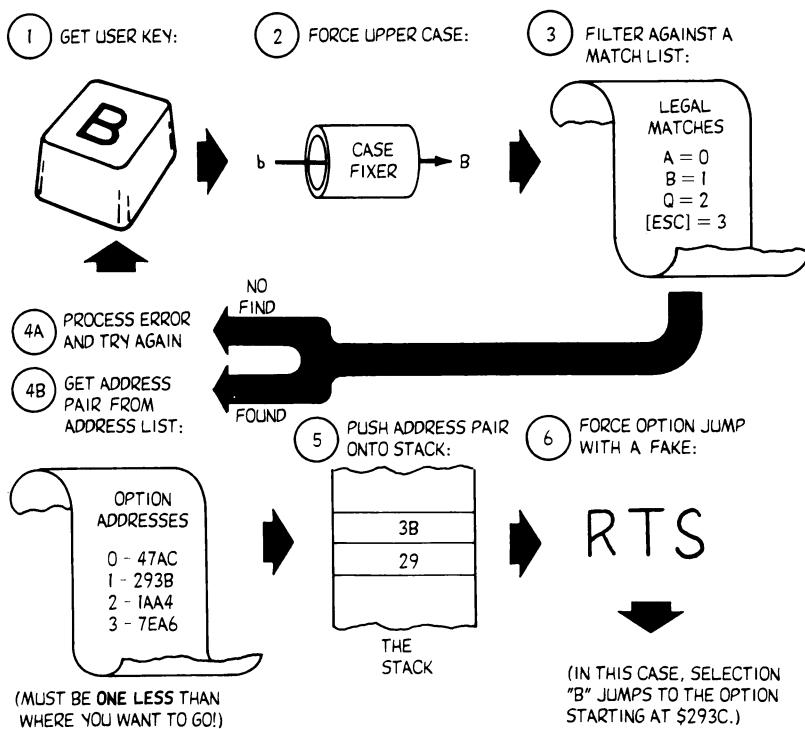
A more subtle, yet far more important, fault of our previous option code is that you have to rewrite the code for each and every use. Sometimes the code is very long. Other times it will be very short. As we have seen time and time again, it is far better to have fixed code that uses changing file values, rather than having to custom rebuild

the code for each and every application. This way, you know your code works ahead of time. Any problems are likely to be file problems that are easily spotted and more easily fixed.

Let's check into the most general and often the "best" way to pick one of many options. To do this, get a character from a program or a keyboard. Then, if needed, change lowercase to uppercase. Next, filter your character by looking into a file to find a character match. If there's no match, process the error and try again. If you do find a match, go to a second file and grab an address to go to. Then, jump to that address.

Something like this . . .

HOW TO PICK AN OPTION:

(1) GET USER KEY:　　(2) FORCE UPPER CASE:　　(3) FILTER AGAINST A MATCH LIST:

LEGAL MATCHES

A = 0
B = 1
Q = 2
[ESC] = 3

CASE FIXER

NO FIND

(4A) PROCESS ERROR AND TRY AGAIN

(4B) GET ADDRESS PAIR FROM ADDRESS LIST:

FOUND

(5) PUSH ADDRESS PAIR ONTO STACK:

(6) FORCE OPTION JUMP WITH A FAKE:

OPTION ADDRESSES

0 - 47AC
1 - 293B
2 - 1AA4
3 - 7EA6

3B
29

THE STACK

RTS

(MUST BE **ONE LESS** THAN WHERE YOU WANT TO GO!)

(IN THIS CASE, SELECTION "B" JUMPS TO THE OPTION STARTING AT $293C.)

There are several distinct parts to a good and flexible option picker. First, you normally will want the same response for a capital letter as for a small one, say for "A" and "a." If you do, you will need *case changer* code. If you are allowing for meaningful, rather than ordered, inputs, then you will also need an *option filter* that converts the selections into a binary file access number.

Then there are possible errors. Sometimes a few *legal* and *expected* responses may all want to go to the *same* option. We can call that jump an *inclusive trap*. The simplest way to handle inclusive traps is

just to repeat the same address in the address file as often as needed. There are ways, of course, to save a byte or two on this, but you end up with custom code if you try this. We will use an error message of "PLEASE TRY ANOTHER LETTER" to show this when it happens in the upcoming GILA demo.

Other times, the input will not match any legal selection. What you have to do here is go get another input since the one you have is no good. You can call this an *exclusive trap*. Exclusive traps should go and try and get another response.

You must always inform the user that you don't like his invalid selection. The trick here is to do it as subtly and gently as possible. More often than not, a brief screen flash or a single speaker click is all you will need. We will use a message of "THAT'S NO LETTER, YOU TURKEY!" in the demo. Naturally, such harshness must be used with discretion in commercial programs.

Should the program, rather than the user, be making the option selection, you will end up in deep trouble if the computer decides to do something it is not set up to do. In *Zork*, machine errors are trapped with a "ZORK INTERNAL ERROR" message. Chances are overwhelming that you have not and will not get one of these *Zork* messages.

Unless you play *Zork* the way I do.

Needless to say, machine errors are never supposed to happen. When and if one does, though, be sure to inform the user that he has just been done in through no fault of his own. It may be a good idea to encourage the user to "close the loop" and contact you personally when this happens.

Not *if*, but *when*.

A final part of the option picker has to actually do a jump or a gosub to the selected option. You have many choices here. Building the jumps into the code as we did above is obviously bad, since the code is no longer general. You can also self-modify your code by having a JMP command whose address you pre-change to the address you want to jump to. This is risky and does not work in ROM, but is cute and compact. You can also force a JSR the same way.

The JMP indirect command is another possibility. Here, you put your address somewhere on page zero, say $06 low and $07 high. Then a JMP ($06) does an indirect jump to your intended address. For a forced subroutine, just do a JSR to an indirect JMP.

But, remember that the original 6502 JMP indirect has a bug in it that prevents you from using it properly on either of the top two bytes of any page. If you relocate your code, or do not watch very carefully where your JMP indirects are, this bug may bomb your code. Page zero real estate is valuable enough that you should go out of your way to avoid using it whenever there are reasonable alternatives.

By the way, certain copy protection fanatics intentionally put their JMP indirects in the "wrong" locations, hoping you miss the turn. The jumped-to locations end up on the bottom of the *same* page, rather than the expected bottom of the *next* page this way. Of course, such childish and inane stunts just add to the fun and challenge of cracking the "uncrackable." Besides, they will bomb on an Apple upgraded to a 65C02. Or on a IIc.

Har har.

Anyway, the way I like to do a jump to an option is with a scheme

called the *forced subroutine return* method. This method is used in the Apple system monitor, so it is not new. But it is super powerful and elegant.

Remember that a subroutine return or RTS checks into the stack and gets the top stack location. It uses this location for the position on the page it is to return to. Then, it goes one deeper into the stack to get the page location. Given the position and the page, the RTS then jumps to this location *plus one.*

Normally, of course, the RTS returns to the code that called it. Now to get sneaky. Take the page address of your option and shove it on the stack with a PHA. Then, take the option position address *minus one* and shove it on the stack with a second PHA. Now, RTS. What happens?

You "return" to the address of your selected option!

Note that two pushes (by you) and two pulls (by the RTS) leave the stack the way it was before you started. So you are still in the same "level" of your code both before and after you force the fake subroutine return. Note also that no page zero locations are committed.

For an earlier and different example of using forced subroutine returns, check back into IMPRINT of Ripoff Module 2.

Let's sum up the parts of our option picker . . .

CASE CHANGER—

Code that forces lowercase letters into their uppercase equivalents.

OPTION FILTER—

Code that finds a match between user inputs and a binary value.

INCLUSIVE TRAP—

Several user selections that all divert to the same option.

EXCLUSIVE TRAP—

Code that finds "illegal" user inputs and suitably handles this type of error.

FORCED SUBROUTINE RETURN—

A JMP indirect that is faked by pushing an address pair onto the stack and then doing an RTS.

Summing up, while there are lots of ways to pick options, we will use a general and powerful file based method that is easy to use and easy to change. It is best suited for six or more unordered choices. The code is very efficient when many different selections are made.

To pick an option, you first change the case of lowercase letters, so that either a capital "A" or a lowercase "a" gets the same response. Then you look for a match in a character file. Finding the match generates a binary number useful as an address pointer.

Should a match be found, the binary number is doubled and used to access an address pair in an address file. This address pair is forced

onto the stack and is then followed by an RTS, doing a jump to the selected option.

Two crucial reminders . . .

When "force feeding" a stack—

ALWAYS push the page address on first, followed by the position address.

ALWAYS use an address ONE LESS than your intended return point.

The sneaky way to automatically remove one from any address is to let your assembler's operand arithmetic handle the chore for you. Thus, instead of a label of TASKA, use TASKA−1 when defining addresses in your address file. The DW command is one good way to handle 2-byte pairs. DW automatically rearranges these pairs into their "position-page" format for you.
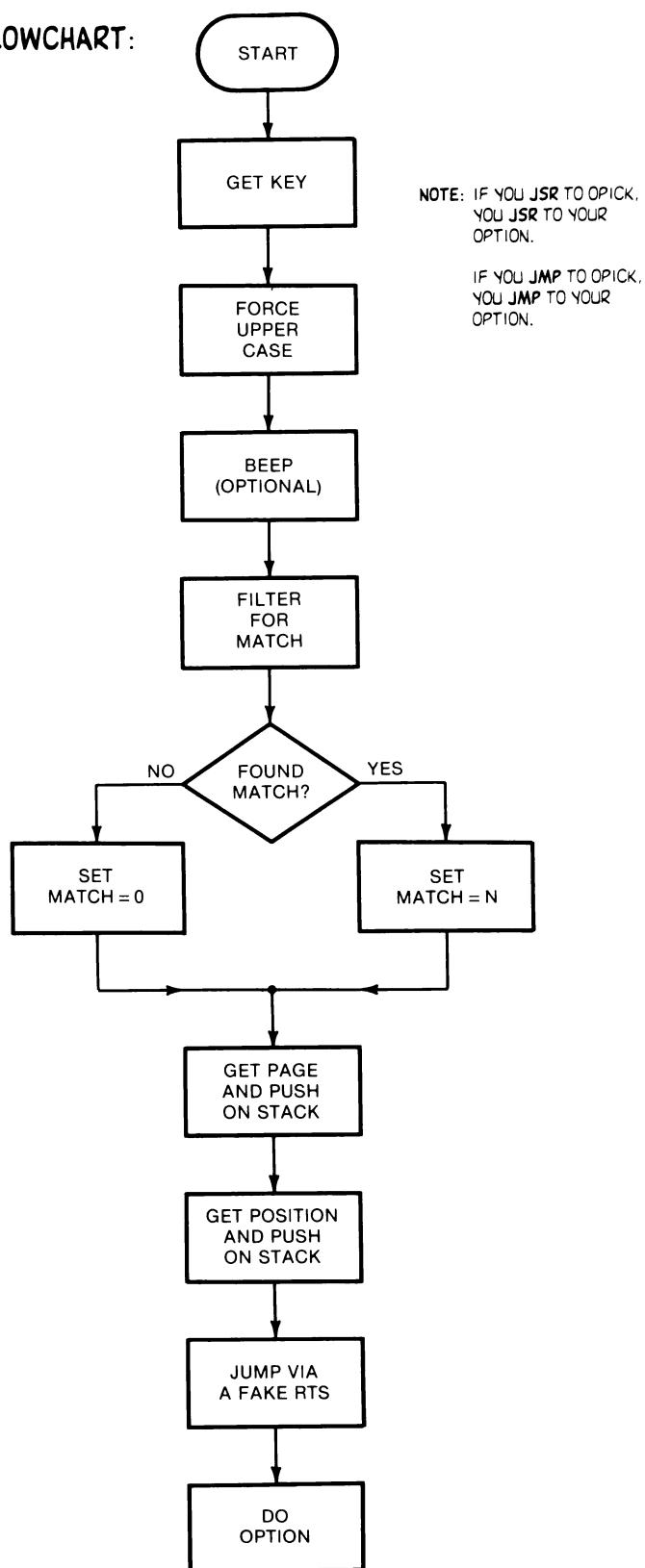
Don't forget these two crucial details: The page goes on the stack first, and RTS ends up one beyond the stack address.

Several matches can point to the same address pair by repeating the address pair when and where needed in the address file. We have seen how this is called an inclusive trap.

Should no match be found, an exclusive trap tries for a new input or generates an error message, informing the user as this happens.

The option picking subroutine is called OPICK. Its flowchart looks like this . . .

OPICK FLOWCHART:

START

GET KEY

FORCE
UPPER
CASE

BEEP
(OPTIONAL)

FILTER
FOR
MATCH

FOUND
MATCH?

NO

YES

SET
MATCH = 0

SET
MATCH = N

GET PAGE
AND PUSH
ON STACK

GET POSITION
AND PUSH
ON STACK

JUMP VIA
A FAKE RTS

DO
OPTION

NOTE: IF YOU JSR TO OPICK,
YOU JSR TO YOUR
OPTION.

IF YOU JMP TO OPICK,
YOU JMP TO YOUR
OPTION.

Some parts of OPICK might not be needed for all uses. For instance, you can delete getting a key if the machine itself is to provide the option selection.

Delete the case changer if you want something different to happen for a capital letter than for a lowercase one. Sometimes, your options will not even be in ASCII. They might be a binary selection. If so, lowercase is meaningless.

The option filter can be deleted if you are certain your option selections are always ordered binary numbers. This is OK for internal use, but, as we've seen, is a poor and unfriendly choice where users are involved.

And, don't use such heavy code for trivial choices. A simple (Y/N) checker can be done much faster with many fewer bytes. For over six choices, the option filter is the better way to go. The more the choices and the more wildly they are arranged, the better the method gets.

The FIXCASE case changer works by testing for a lowercase ASCII letter. If it gets one of these, then $20 is subtracted as needed to get uppercase. For instance, a lowercase "a" is an ASCII $E1. Subtract $20 to get $C1, the ASCII uppercase "A." It pays to test for "z" as well as "a" so that any punctuation above ASCII $FA does not get changed. We've shown high ASCII here, as you get off the Apple keyboard *before* resetting the keystroke.

Any match character can go in any order, except that the position of the address in the address table must be exactly *twice* the position of the match character in the match file. All this says is that the match must line up with where you want the match to go to. The doubling is needed for the 2-byte absolute address *pairs* and is handled with an ASL multiplier.

The matches in the match file can go in any order. The obvious and cleanest arrangement is to put the selections in logical user input order. Another way is to put the addresses in the order they appear in the program. Still another way is to put the often used matches first, in an attempt to gain a slight speedup. Just be sure that the match and the match address are aligned to each other.

We have put the match values in alphabetical order. Once again, though, you can put any mix of numbers, letters, and control characters in any order, skipping around anywhere you like.

As we have seen, the cleanest way to handle inclusive traps is to repeat the address pair as often as needed in your address file.

We used an inverse title for this demo, like we did with IMPRINT and FLPRINT. These titles are quick and dirty to do, but they are usually far too garish to use in a commercial program. A single inverse line cuts the tops of uppercase letters and random tops and bottoms of lowercase. The obvious cure for this of using three inverse lines to form a box usually is too "loud" for the rest of the screen.

So, do as I say, not as I do . . .

AVOID using inverse text headers and titles on commercial programs.

These are too garish and imperfect to give you acceptable results.

You have a choice of using your options as subroutines or else as same-level jumps. If, as we did in GILA, you JSR to OPICK and then force-return to your option, an RTS at the end of the option returns you to the code that is calling OPICK.

On the other hand, you could JMP to OPICK and then force-return to your option. Here, a JMP at the end of the option is needed to return you to a calling code.

In one method, the options are subroutines. In the other, they are at the same level as the code that calls OPICK.

To adapt OPICK to your own needs, just change the MAXMATCH number to equal the total number of options, change the MATCHFL file to hold the characters you are matching against, and change the JMPFLE to hold the addresses you want to jump to. A reassembly, of course, will be needed. As usual, labels that name each option you are to jump to greatly simplify and automate the process of building this file. Makes it fun even.

Time for . . .

A Demo

Normally, your option picker will jump to lots of wildly different types of code in your main program. To keep DEMO6 simple, we will still jump to lots of different points in the main program, but the action at each option point will be rather simple and sort of redundant. Now, there probably are better ways to write a program that does what this demo does, but, remember that we are trying to show the *method* of using an option picker to go many places in a larger program.

DEMO6 is our demo, and what it does is generate the name of a town in exchange for a user input that matches the first letter of that town. Once you have recovered from the initial excitement of such a stupendous program, look carefully to see how the options each go to a selected code module, and how the inclusive and exclusive traps are working. Note how both control commands and letter inputs are handled. See how the ESC key exits the program for you.

Incidentally, we've used our own key getter, rather than GETKEY. It's tricky to handle escape commands with GETKEY, and GETKEY gives slightly different results on a II versus a IIe. The prompt gets entered by using IMPRINT to print a prompt, followed by a backspace, followed by the double zero exit.

DEMO6 has borrowed the IMPRINT code from Ripoff Module 2, so be sure that either this code, a copy of it, or else a copy of THE WHOLE BALL OF WAX is present in the machine.

MIND BENDERS

—Rework the demo to use your own towns in your own local area.

—Change the demo to other topics, such as autos, aircraft, animals, vegetables, or nurflongs.

—What other uses are there for the forced subroutine method?

—How long does the option picker take to process an option?

—Show how to use your options as same level code, rather than as subroutines.

—What other user prompting can be used in place of the time delay?

—Explain away those two PLAs in the exit code. Why are they used?

—Try to BRUN OPTION PICKER directly from your disk, and [ESC] will not exit you from your program. Why?

—Display a different LORES or HIRES picture for each selection, along with suitable sound.

```
                        PROGRAM RM-6
                            OPTION PICKER
```

```
----- NEXT OBJECT FILE NAME IS OPTION PICKER
6C00:                3              ORG   $6C00      ; PUT MODULE #6 AT $6C00


6C00:                5 ;    ******************************************
6C00:                6 ;    *                                        *
6C00:                7 ;    *              -< OPTION PICKER >-        *
6C00:                8 ;    *                                        *
6C00:                9 ;    *     ( JUMPING SIX WAYS FROM SUNDAY )    *
6C00:               10 ;    *                                        *
6C00:               11 ;    *        VERSION 1.0   ($6C00-$6EDD)      *
6C00:               12 ;    *                                        *
6C00:               13 ;    *                5-24-83                  *
6C00:               14 ;    *........................................*
6C00:               15 ;    *                                        *
6C00:               16 ;    *          COPYRIGHT C 1983 BY           *
6C00:               17 ;    *                                        *
6C00:               18 ;    *     DON LANCASTER AND SYNERGETICS       *
6C00:               19 ;    *     BOX 1300, THATCHER AZ., 85552       *
6C00:               20 ;    *                                        *
6C00:               21 ;    *     ALL COMMERCIAL RIGHTS RESERVED      *
6C00:               22 ;    *                                        *
6C00:               23 ;    ******************************************


6C00:               25 ;            *** WHAT IT DOES ***

6C00:               27 ;    THIS MODULE SHOWS YOU HOW TO JUMP TO ONE OF MANY
6C00:               28 ;    POSSIBLE POINTS TO CONTINUE RUNNING A PROGRAM.
6C00:               29 ;



6C00:               31 ;            *** HOW TO USE IT ***

6C00:               33 ;    TO USE THE OPTION PICKER:
6C00:               34 ;
6C00:               35 ;       REASSEMBLE WITH YOUR NUMBER OF MATCHES  IN MATCHN,
6C00:               36 ;       YOUR MATCHES IN MATCHFL AND YOUR JUMP
6C00:               37 ;       VECTORS MINUS ONE IN JMPFL.   THEN JSR
6C00:               38 ;       OPICK AT $6E45 (28229).


6C00:               40 ;    TO RUN THE GILA TOWNS DEMO:
6C00:               41 ;
6C00:               42 ;       JSR GILA AT $6C00 OR CALL 27648.
```

PROGRAM RM-6, CONT'D . . .

```
6C00:              45 ;              *** GOTCHAS ***

6C00:              47 ;   THE IMPRINT SUBROUTINE MUST BE PRESENT IN THE
6C00:              48 ;   MACHINE.  PRELOAD "IMPRINT" OR "THE WHOLE BALL
6C00:              49 ;   OF WAX" TO DO THIS.
6C00:              50 ;
6C00:              51 ;   JUMP VECTORS MUST BE IN THE USUAL "POSITION-
6C00:              52 ;   PAGE" ORDER.  THE ORDER IN MATCHFL MUST EQUAL
6C00:              53 ;   THE ORDER IN TASKFL.
6C00:              54 ;   JUMP VECTORS MUST BE ONE LESS THAN
6C00:              55 ;   THEIR ACTUAL RETURN POINTS!


6C00:              57 ;              *** ENHANCEMENTS ***

6C00:              59 ;   MATCHED CHARACTERS CAN BE IN ANY ORDER AND MAY
6C00:              60 ;   INCLUDE CONTROL CHARACTERS.
6C00:              61 ;
6C00:              62 ;   INCLUSIVE TRAPS ARE DONE BY REPEATING THE TASKFL
6C00:              63 ;   JUMP VECTORS AS OFTEN AS NEEDED.
6C00:              64 ;


6C00:              66 ;              *** RANDOM COMMENTS ***

6C00:              68 ;   THERE ARE CERTAINLY BETTER WAYS TO HANDLE THE
6C00:              69 ;   GILA TOWNS DEMO THAN THIS.  IN REAL LIFE, EACH
6C00:              70 ;   "TOWN" REPRESENTS A DIFFERENT AND UNIQUE HIGH
6C00:              71 ;   LEVEL PROGRAMMING TASK.
6C00:              72 ;
6C00:              73 ;   THE DEMO ALSO SHOWS HOW TO CHANGE THE SCROLLING
6C00:              74 ;   TEXT WINDOW UNDER PROGRAM CONTROL.
```

PROGRAM RM-6, CONT'D . . .

```
6C00:          77 ;              *** HOOKS ***


FC58:          79 HOME     EQU  $FC58    ; CLEAR TEXT SCREEN AND HOME CURSOR
FB2F:          80 INIT     EQU  $FB2F    ; INITIALIZE TEXT SCREEN
C000:          81 IOADR    EQU  $C000    ; KEYBOARD INPUT LOCATION
C010:          82 KBDSTRB  EQU  $C010    ; KEYBOARD STROBE RESET
FE80:          83 SETINV   EQU  $FE80    ; SET INVERSE SCREEN
FE84:          84 SETNORM  EQU  $FE84    ; SET NORMAL SCREEN
C030:          85 SPKR     EQU  $C030    ; SPEAKER CLICK OUTPUT
FCA8:          86 WAIT     EQU  $FCA8    ; MONITOR TIME DELAY


666B:          88 IMPRINT  EQU  $666B    ; LINK TO IMPRINT SUBROUTINE


0020:          90 WNDLFT   EQU  $20      ; LEFT SIDE OF SCROLL WINDOW
0021:          91 WNDWTH   EQU  $21      ; WIDTH OF SCROLL WINDOW
0022:          92 WNDTOP   EQU  $22      ; TOP OF SCROLL WINDOW
0023:          93 WNDBTM   EQU  $23      ; BOTTOM OF SCROLL WINDOW
0024:          94 CH       EQU  $24      ; CURSOR HORIZONTAL POSITION
0025:          95 CV       EQU  $25      ; CURSOR VERTICAL POSITION
0033:          96 PROMPT   EQU  $33      ; PROMPT SYMBOL


6C00:          98 ;              *** TEXTFILE COMMANDS ***


0088:         100 B        EQU  $88      ; BACKSPACE
008D:         101 C        EQU  $8D      ; CARRIAGE RETURN
0084:         102 D        EQU  $84      ; DOS ATTENTION
009B:         103 E        EQU  $9B      ; ESCAPE
008A:         104 L        EQU  $8A      ; LINEFEED
0060:         105 P        EQU  $60      ; FLASHING PROMPT
0000:         106 X        EQU  $00      ; END OF MESSAGE
```

PROGRAM RM-6, CONT'D . . .

```
6C00:            109 ;         *** GILA TOWNS DEMO ***
6C00:            110 ;
6C00:            111 ;    THIS PROGRAM EXERCISES THE OPTION
6C00:            112 ;    PICKER SUBROUTINE OPICK.
6C00:            113 ;
6C00:            114 ;    EACH "TOWN" REPRESENTS A DIFFERENT
6C00:            115 ;    HIGH LEVEL PROGRAM TASK.


6C00:20 2F FB   117 GILA    JSR  INIT      ; SET UP TEXT SCREEN
6C03:20 58 FC   118         JSR  HOME      ; CLEAR SCREEN AND HOME CURSOR
6C06:A9 07      119         LDA  #07       ; TAB 7 TO RIGHT
6C08:85 24      120         STA  CH        ;
6C0A:20 80 FE   121         JSR  SETINV    ; INVERSE TITLE
6C0D:20 6B 66   122         JSR  IMPRINT   ; PUT DOWN TITLE
6C10:8A 8A 8A   123         DFB           L,L,L
6C13:CF D0 D4   124         ASC            "OPTION PICKER DEMO"
6C16:C9 CF CE
6C19:A0 D0 C9
6C1C:C3 CB C5
6C1F:D2 A0 C4
6C22:C5 CD CF
6C25:8D 8D 8D   125         DFB           C,C,C,C,X
6C28:8D 00

6C2A:20 84 FE   127         JSR  SETNORM   ; BACK TO NORMAL TEXT
6C2D:20 6B 66   128         JSR  IMPRINT   ; PUT DOWN INSTRUCTIONS
6C30:D4 D9 D0   129         ASC            "TYPE THE FIRST LETTER TO GET THE"
6C33:C5 A0 D4
6C36:C8 C5 A0
6C39:C6 C9 D2
6C3C:D3 D4 A0
6C3F:CC C5 D4
6C42:D4 C5 D2
6C45:A0 D4 CF
6C48:A0 C7 C5
6C4B:D4 A0 D4
6C4E:C8 C5
6C50:8D       130         DFB           C
6C51:C6 D5 CC  131         ASC            "FULL NAME OF A GILA VALLEY TOWN:"
6C54:CC A0 CE
6C57:C1 CD C5
6C5A:A0 CF C6
6C5D:A0 C1 A0
6C60:C7 C9 CC
6C63:C1 A0 D6
6C66:C1 CC CC
6C69:C5 D9 A0
6C6C:D4 CF D7
6C6F:CE BA
```

PROGRAM RM-6, CONT'D . . .

```
6C71:8D 8D 8D  134          DFB          C,C,C,C
6C74:8D
6C75:A0 A0 A0  135          ASC          "        --->  "
6C78:A0 A0 A0
6C7B:A0 A0 AD
6C7E:AD AD BE
6C81:A0
6C82:8D 8D 8D  136          DFB          C,C,C,C,C,C
6C85:8D 8D 8D
6C88:A0 A0 A0  137          ASC          /        (USE "ESC" TO EXIT)/
6C8B:A0 A0 A0
6C8E:A8 D5 D3
6C91:C5 A0 A2
6C94:C5 D3 C3
6C97:A2 A0 D4
6C9A:CF A0 C5
6C9D:D8 C9 D4
6CA0:A9
6CA1:00        138          DFB          X

6CA2:A9 0D     140          LDA  #$0D    ; SET TIGHT WINDOW
6CA4:85 20     141          STA  WNDLFT  ;
6CA6:A9 15     142          LDA  #$15    ;
6CA8:85 21     143          STA  WNDWTH  ;
6CAA:A9 0C     144          LDA  #$0C    ;
6CAC:85 22     145          STA  WNDTOP  ;
6CAE:A9 0F     146          LDA  #$0F    ;
6CB0:85 23     147          STA  WNDBTM  ;
6CB2:20 58 FC  148          JSR  HOME    ; GET IN WINDOW
6CB5:A9 60     149          LDA  #P      ; CHANGE PROMPT
6CB7:85 33     150          STA  PROMPT  ;

6CB9:20 6B 66  152 DOOPT    JSR  IMPRINT ; ADD WINKING CURSOR
6CBC:60 88 00  153          DFB          P,B,X
6CBF:20 44 6E  154          JSR  OPICK   ; GET AND DO OPTIONS AS SUBS

6CC2:A2 0D     156 CONT6    LDX  #13     ; MOST OPTIONS RETURN TO HERE
6CC4:20 A8 FC  157 STALL6   JSR  WAIT    ; STALL FOR DISPLAY TIME
6CC7:CA        158          DEX          ;
6CC8:D0 FA     159          BNE  STALL6  ;

6CCA:20 58 FC  161          JSR  HOME    ; ERASE OLD SCREEN
6CCD:A2 28     162          LDX  #$28    ;
6CCF:20 7B 6E  163          JSR  QUIP    ; BLORK
6CD2:4C B9 6C  164          JMP  DOOPT   ; AND REPEAT
```

PROGRAM RM-6, CONT'D . . .

```
6CD5:              167 ;       *** THE ACTUAL TASKS ***


6CD5:20 6B 66      169 TASKA   JSR  IMPRINT;  TASK A
6CD8:C1 D2 D4      170         ASC            "ARTESIA"
6CDB:C5 D3 C9
6CDE:C1
6CDF:00            171         DFB            X
6CEC:60            172         RTS            ;


6CE1:20 6B 66      174 TASKB   JSR  IMPRINT;  TASK B
6CE4:C2 CF CE      175         ASC            "BONITA"
6CE7:C9 D4 C1
6CEA:00            176         DFB            X
6CEB:60            177         RTS            ;


6CEC:20 6B 66      179 TASKC   JSR  IMPRINT;  TASK C
6CEF:C3 CC C9      180         ASC            "CLIFTON"
6CF2:C6 D4 CF
6CF5:CE
6CF6:00            181         DFB            X
6CF7:60            182         RTS            ;


6CF8:20 6B 66      184 TASKD   JSR  IMPRINT;  TASK D
6CFB:C4 D5 CE      185         ASC            "DUNCAN"
6CFE:C3 C1 CE
6D01:00            186         DFB            X
6D02:60            187         RTS            ;


6D03:20 6B 66      189 TASKE   JSR  IMPRINT;  TASK E
6D06:C5 C4 C5      190         ASC            "EDEN"
6D09:CE
6D0A:00            191         DFB            X
6D0B:60            192         RTS            ;


6D0C:20 6B 66      194 TASKF   JSR  IMPRINT;  TASK F
6D0F:C6 D2 C1      195         ASC            "FRANKLIN"
6D12:CE CB CC
6D15:C9 CE
6D17:00            196         DFB            X
6D18:60            197         RTS            ;
```

PROGRAM RM-6, CONT'D . . .

```
6D19:20 6B 66   200 TASKG   JSR   IMPRINT;  TASK G
6D1C:C7 D5 D4   201         ASC             "GUTHRIE"
6D1F:C8 D2 C9
6D22:C5
6D23:00         202         DFB             X
6D24:60         203         RTS             ;


6D25:20 6B 66   205 TASKH   JSR   IMPRINT;  TASK H
6D28:C8 C5 CC   206         ASC             "HELIOGRAPH"
6D2B:C9 CF C7
6D2E:D2 C1 D0
6D31:C8
6D32:00         207         DFB             X
6D33:60         208         RTS             ;


6D34:20 6B 66   210 TASKI   JSR   IMPRINT;  TASK I
6D37:C9 CE C4   211         ASC             "INDIAN SPRINGS"
6D3A:C9 C1 CE
6D3D:A0 D3 D0
6D40:D2 C9 CE
6D43:C7 D3
6D45:00         212         DFB             X
6D46:60         213         RTS             ;


6D47:20 6B 66   215 TASKJ   JSR   IMPRINT;  TASK J
6D4A:CA C1 C3   216         ASC             "JACKSON ESTATES"
6D4D:CB D3 CF
6D50:CE A0 C5
6D53:D3 D4 C1
6D56:D4 C5 D3
6D59:00         217         DFB             X
6D5A:60         218         RTS             ;


6D5B:20 6B 66   220 TASKK   JSR   IMPRINT;  TASK K
6D5E:CB CC CF   221         ASC             "KLONDYKE"
6D61:CE C4 D9
6D64:CB C5
6D66:00         222         DFB             X
6D67:60         223         RTS             ;


6D68:20 6B 66   225 TASKL   JSR   IMPRINT;  TASK L
6D6B:CC C9 D4   226         ASC             "LITTLE TULSA"
6D6E:D4 CC C5
6D71:A0 D4 D5
6D74:CC D3 C1
6D77:00         227         DFB             X
6D78:60         228         RTS             ;
```

PROGRAM RM-6, CONT'D . . .

```
6D79:20 6B 66   231 TASKM    JSR   IMPRINT;   TASK M
6D7C:CD CF D2   232          ASC              "MORENCI"
6D7F:C5 CE C3
6D82:C9
6D83:00         233          DFB              X
6D84:60         234          RTS              ;


6D85:20 6B 66   236 TASKN    JSR   IMPRINT;   TASK N
6D88:CE C1 C3   237          ASC              "NACHES"
6D8B:C8 C5 D3
6D8E:00         238          DFB              X
6D8F:60         239          RTS              ;


6D90:           241 ;              TASK O DEFAULTS TO INCTRAP


6D90:20 6B 66   243 TASKP    JSR   IMPRINT;   TASK P
6D93:D0 C9 CD   244          ASC              "PIMA"
6D96:C1
6D97:00         245          DFB              X
6D98:60         246          RTS              ;


6D99:           248 ;              TASK Q DEFAULTS TO INCTRAP


6D99:20 6B 66   250 TASKR    JSR   IMPRINT;   TASK R
6D9C:D2 CF D0   251          ASC              "ROPER LAKE"
6D9F:C5 D2 A0
6DA2:CC C1 CB
6DA5:C5
6DA6:00         252          DFB              X
6DA7:60         253          RTS              ;


6DA8:20 6B 66   255 TASKS    JSR   IMPRINT;   TASK S
6DAB:D3 C1 C6   256          ASC              "SAFFORD"
6DAE:C6 CF D2
6DB1:C4
6DB2:00         257          DFB              X
6DB3:60         258          RTS              ;


6DB4:20 6B 66   260 TASKT    JSR   IMPRINT;   TASK T
6DB7:D4 C8 C1   261          ASC              "THATCHER"
6DBA:D4 C3 C8
6DBD:C5 D2
6DBF:00         262          DFB              X
6DC0:60         263          RTS              ;


6DC1:           265 ;              TASK U DEFAULTS TO INCTRAP
```

PROGRAM RM-6, CONT'D . . .

```
6DC1:20 6B 66    268 TASKV    JSR   IMPRINT;   TASK V
6DC4:D6 C9 D2    269          ASC             "VIRDEN"
6DC7:C4 C5 CE
6DCA:00          270          DFB              X
6DCB:60          271          RTS              ;


6DCC:20 6B 66    273 TASKW    JSR   IMPRINT;   TASK W
6DCF:D7 C8 C9    274          ASC             "WHITLOCK CIENEGA"
6DD2:D4 CC CF
6DD5:C3 CB A0
6DD8:C3 C9 C5
6DDB:CE C5 C7
6DDE:C1
6DDF:00          275          DFB              X
6DE0:60          276          RTS              ;


6DE1:            278 ;        TASK X DEFAULTS TO INCTRAP


6DE1:20 6B 66    280 TASKY    JSR   IMPRINT;   TASK Y
6DE4:D9 CF D2    281          ASC             "YORK VALLEY"
6DE7:CB A0 D6
6DEA:C1 CC CC
6DED:C5 D9
6DEF:00          282          DFB              X
6DF0:60          283          RTS              ;


6DF1:            285 ;        TASK Z DEFAULTS TO INCTRAP
```

PROGRAM RM-6, CONT'D . . .

```
6DF1:20 6B 66   288 INCTRAP JSR   IMPRINT;   INCLUSIVE DEFAULT TRAP
6DF4:D3 CF D2   289         ASC              "SORRY, PLEASE TRY"
6DF7:D2 D9 AC
6DFA:A0 D0 CC
6DFD:C5 C1 D3
6E00:C5 A0 D4
6E03:D2 D9
6E05:8D         290         DFB              C
6E06:D3 CF CD   291         ASC              "SOME OTHER LETTER"
6E09:C5 A0 CF
6E0C:D4 C8 C5
6E0F:D2 A0 CC
6E12:C5 D4 D4
6E15:C5 D2
6E17:00         292         DFB              X
6E18:60         293         RTS              ;


6E19:20 6B 66   295 ERRTRAP JSR   IMPRINT;   ILLEGAL KEY DEFAULT
6E1C:D4 C8 C1   296         ASC              "THATS NO LETTER
6E1F:D4 D3 A0
6E22:CE CF A0
6E25:CC C5 D4
6E28:D4 C5 D2
6E2B:8D         297         DFB              C
6E2C:A0 A0 D9   298         ASC              "   YOU TURKEY!
6E2F:CF D5 A0
6E32:D4 D5 D2
6E35:CB C5 D9
6E38:A1
6E39:00         299         DFB              X
6E3A:60         300         RTS              ;


6E3B:20 2F FB   302 QUIT6   JSR   INIT       ; RESTORE NORMAL TEXT WINDOW
6E3E:20 58 FC   303         JSR   HOME       ; HOME CURSOR AND CLEAR SCREEN
6E41:68         304         PLA              ; BYPASS GILA; GO STRAIGHT
6E42:68         305         PLA              ; TO MONITOR OR CALLING CODE
6E43:60         306         RTS              ; FOR COMPLETE EXIT
```

PROGRAM RM-6, CONT'D . . .

```
6E44:            309 ;            *** OPTION PICKER SUBROUTINE ***
6E44:            310 ;
6E44:            311 ;    FOR OTHER USES, THIS SUB HAS TO BE LINKED TO
6E44:            312 ;    YOUR OWN MATCHN MATCH NUMBER, YOUR MATCHF
6E44:            313 ;    CHARACTER MATCHER FILE AND YOUR JMPFLE VECTORS.
6E44:            314 ;


6E44:2C 10 C0    316 OPICK    BIT   KBDSTRB   ; LOCK OUT EARLY HITS
6E47:AD 00 C0    317 LOOK6    LDA   IOADR     ; GET KEY.  CAN'T USE KEYIN
6E4A:10 FB       318          BPL   LOOK6     ;  BECAUSE WE NEED ESC COMMAND.
6E4C:2C 10 C0    319          BIT   KBDSTRB   ; RESET STROBE

6E4F:2C 6F 6E    321          JSR   FIXCASE   ; FORCE UPPERCASE

6E52:A2 0A       323          LDX   #10
6E54:20 7B 6E    324          JSR   QUIP      ; BLORK

6E57:AE 8A 6E    326          LDX   MATCHN    ; GET LEGAL NUMBER OF MATCHES
6E5A:DD 8B 6E    327 SCAN6    CMP   MATCHFL,X ; SEARCH FOR A MATCH
6E5D:F0 03       328          BEQ   GOTMTCH   ; FOUND
6E5F:CA          329          DEX             ; TRY NEXT
6E60:10 F8       330          BPL   SCAN6;

6E62:E8          332 GOTMTCH  INX             ; MAKES ZERO A MISS
6E63:8A          333          TXA             ; GET JUMP VECTOR
6E64:0A          334          ASL   A         ; DOUBLE POINTER
6E65:AA          335          TAX             ;
6E66:BD A7 6E    336          LDA   JMPFL+1,X ; GET PAGE ADDRESS FIRST!
6E69:48          337          PHA             ;  AND FORCE ON STACK
6E6A:BD A6 6E    338          LDA   JMPFL,X   ; GET POSITION ADDRESS
6E6D:48          339          PHA             ;   AND FORCE ON STACK
6E6E:60          340          RTS             ; JUMP VIA FORCED SUBROUTINE RETURN
```

PROGRAM RM-6, CONT'D . . .

```
6E6F:            343 ;          *** CASE FIXER SUBROUTINE ***


6E6F:            345 ;   TESTS THE ACCUMULATOR FOR A LOWERCASE
6E6F:            346 ;   CHARACTER.  IF PRESENT, FORCES UPPERCASE
6E6F:            347 ;   BY ADDING $20.  USES HIGH ASCII.


6E6F:C9 E1       349 FIXCASE CMP  #$E1      ; IF "a" OR MORE
6E71:90 07       350         BCC  NOFIX6    ;
6E73:C9 FB       351         CMP  #$FB      ;  AND IF "z" OR LESS
6E75:B0 03       352         BCS  NOFIX6    ;
6E77:38          353         SEC            ;  THEN SUBTRACT $20 TO
6E78:E9 20       354         SBC  #$20      ;  FORCE UPPER CASE
6E7A:60          355 NOFIX6  RTS            ;  AND RETURN




6E7B:            357 ;          *** QUIP SUBROUTINE ***
6E7B:            358 ;
6E7B:            359 ;   MAKES NOISE. X SETS THE PITCH.  THE
6E7B:            360 ;    PITCH IS PROPORTIONAL TO THE DURATION.
6E7B:            361 ;    WHICH IS OK FOR THIS SIMPLE USE BUT
6E7B:            362 ;    SHOULD BE AVOIDED MOST EVERYWHERE ELSE.


6E7B:48          364 QUIP    PHA            ; SAVE ACCUMULATOR
6E7C:A0 3C       365         LDY  #60       ; NUMBER OF CYCLES
6E7E:8A          366 NXT6    TXA            ; PITCH
6E7F:2C 30 C0    367         BIT  SPKR      ; WHAP SPEAKER
6E82:20 A8 FC    368         JSR  WAIT      ;
6E85:88          369         DEY            ; NEXT CYCLE
6E86:D0 F6       370         BNE  NXT6      ;
6E88:68          371         PLA            ; RESTORE ACCUMULATOR
6E89:60          372         RTS            ; AND EXIT
```

PROGRAM RM-6, CONT'D . . .

```
6E8A:              375 ;         *** OPTION PICKER FILES ***


6E8A:              377 ;         MATCHN HOLDS THE NUMBER OF MATCHES.
6E8A:              378 ;         MATCHFL HOLDS THE LEGAL CHARACTERS.
6E8A:              379 ;         JUMPFL HOLDS THE JUMP VECTORS.
6E8A:              380 ;
6E8A:              381 ;         NOTE THAT ANY NUMBER OF CHARACTERS
6E8A:              382 ;         AND CONTROL COMMANDS MAY BE USED
6E8A:              383 ;         IN ANY ORDER, BUT THAT EACH MUST
6E8A:              384 ;         POSITION MATCH ITS JUMPFL VECTOR.


6E8A:1B            386 MATCHN  DFB   27          ; NUMBER OF LEGAL MATCHES GOES HERE


6E8B:9B            388 MATCHFL DFB   E           ; FOR ESCAPE

6E8C:C1 C2 C3      390         ASC               "ABCDEFGHIJKLM"
6E8F:C4 C5 C6
6E92:C7 C8 C9
6E95:CA CB CC
6E98:CD

6E99:CE CF D0      392         ASC               "NOPQRSTUVWXYZ"
6E9C:D1 D2 D3
6E9F:D4 D5 D6
6EA2:D7 D8 D9
6EA5:DA


6EA6:18 6E         394 JMPFL   DW    ERRTRAP-1 ; NOT A LEGAL KEY
6EA8:3A 6E         395         DW    QUIT6-1   ; EXIT ON ESCAPE
6EAA:D4 6C         396         DW    TASKA-1   ; DO LETTERED TASK
6EAC:E0 6C         397         DW    TASKB-1   ;
6EAE:EB 6C         398         DW    TASKC-1   ;
6EB0:F7 6C         399         DW    TASKD-1   ;
6EB2:02 6D         400         DW    TASKE-1   ;
6EB4:0B 6D         401         DW    TASKF-1   ;
6EB6:18 6D         402         DW    TASKG-1   ;
6EB8:24 6D         403         DW    TASKH-1   ;
6EBA:33 6D         404         DW    TASKI-1   ;
6EBC:46 6D         405         DW    TASKJ-1   ;
6EBE:5A 6D         406         DW    TASKK-1   ;
6EC0:67 6D         407         DW    TASKL-1   ;
6EC2:78 6D         408         DW    TASKM-1   ;
6EC4:84 6D         409         DW    TASKN-1   ;
```

PROGRAM RM-6, CONT'D . . .

```
6EC6:F0 6D     412          DW     INCTRAP-1 ; LEGAL BUT NO TOWN
6EC8:8F 6D     413          DW     TASKP-1   ;
6ECA:F0 6D     414          DW     INCTRAP-1 ; LEGAL BUT NO TOWN
6ECC:98 6D     415          DW     TASKR-1   ;
6ECE:A7 6D     416          DW     TASKS-1   ;
6ED0:B3 6D     417          DW     TASKT-1   ;
6ED2:F0 6D     418          DW     INCTRAP-1 ; LEGAL BUT NO TOWN
6ED4:C0 6D     419          DW     TASKV-1   ;
6ED6:CB 6D     420          DW     TASKW-1   ;
6ED8:F0 6D     421          DW     INCTRAP-1 ; LEGAL BUT NO TOWN
6EDA:E0 6D     422          DW     TASKY-1   ;
6EDC:F0 6D     423          DW     INCTRAP-1 ; LEGAL BUT NO TOWN
```

*** SUCCESSFUL ASSEMBLY: NO ERRORS

RANDOM NUMBERS

pseudo-random number generator is fast, flexible, and free of defects

Random numbers are essential for many computer uses, from the throw of a die, through animated game motions, to industrial simulations. How can you introduce randomness into your programs?

It turns out that there are two types of "random" numbers. A *real* random number is a number that can be one of many equally likely values. A *pseudo-random* number is the next number available in a contrived series that appears on the surface to be any one of many equally likely values . . .

> **RANDOM NUMBER—**
>
> **A number that can assume any one of many equally likely values.**
>
> **PSEUDO-RANDOM NUMBER—**
>
> **The next number available in a contrived series that appears on the surface to be any one of many equally likely values.**

The advantages of "real" random numbers is that they are truly

unpredictable. Disadvantages of real random numbers include that they are hard or inconvenient to generate and that there is no way to get the same random sequence back over again at a later time.

There is a very simple and very useful real random number generator built into your Apple. Any time you use the monitor subroutine KEYIN, there is a 16-bit counter involving locations RNDL and RNDH that gets incremented a random number of times. The randomness comes about since there is no control over how long a user waits between keystrokes. RNDL is located at $4E and RNDH is located at $4F on page zero. The monitor routines GETLN, GETLNZ, GETLN1, RDCHAR, and RDKEY all use KEYIN, so any of these can be used to fetch a new random number . . .

> **To generate a real random number with your Apple, use the monitor routine KEYIN and then read the 16-bit true random result at $4E and $4F.**

The result is a truly random 16-bit number every time. For a new random number, have the user make repeated use of KEYIN, such as with a "HIT ANY KEY TO CONTINUE," or even start out with the flea-bitten "HI, WHAT'S YOUR NAME?" prompt.

If you don't need the full 16 bits, just mask off those you do want. One bit gives you a yes-no decision. A pair of bits generates the random digits from 0–3 and so on. For a RND(6), do a RND(8) instead, and, if you get a 6 or 7 result, go fish again. Or, better yet, you could also write your own version of GETKEY that counts your own base six counter round and round.

Same goes for any other modulo.

By modulo, we mean . . .

> **MODULO—**
>
> **The "N" in RND (N).**
>
> **Note that RND(N) returns with one of N possible values, ranging from ZERO to ONE LESS THAN N.**

Uh, better repeat that. The modulo is the total number of different random numbers you can get back. Since zero is always one of them, the range of numbers will go from zero to $N - 1$.

You never get a value of N for RND(N).

At any rate, using RNDL and RNDH, or else your own software counter for true randomness is very simple. But, there are at least two big disadvantages.

First and worst, the user must hit a key for *every* new random number you need. This gets old fast if more than a dozen selections are involved. Sometimes you can disguise what's happening in a game where lots of keystrokes are involved, but not often.

Secondly, this is a slow process that takes many milliseconds. You can generate pseudo-random numbers hundreds or even thousands of times faster.

And, finally, there is no way to get the same random numbers back again in the same sequence, for replays, or for "noise that repeats."

So, while you have a true random number generator in your Apple and while it is very simple to access, you may not be able to do very much with it.

What About Applesoft's RND?

The advantages of pseudo-random number sequences are that they are easy to generate, and you can easily get the same short and apparently "random" sequence back as often as you like. This is handy for replaying a hand of cards, or to provide "noise that repeats" for industrial testing. You can also do this much faster than you can waiting for someone to press a key.

Applesloth has a subroutine in it that is a failed attempt at pseudo-random number generation.

By now, just about everyone knows that there is a fatal flaw in the Applesloth random number generator, that causes things to repeat in an annoying and frustratingly short way. And, no, the published fixes don't help enough to be useful. So, besides it taking forever to generate a random number, this subroutine simply does not work . . .

> **APPLESOFT RND AIN'T.**
> **DON'T USE IT!**

The fundamental problem is twofold. First, and more or less fixable, the Applesloth RND function does not "reseed" itself every time. The published repairs help this bunches, by using RNDL and RNDH as seeds.

Secondly, and fatally, any pseudo-random sequence generator is supposed to work by making the sequence so long that the numbers will apparently "never" repeat. For many argument values, the Applesloth RND generator does in fact generate an acceptably long sequence. But there are some exactly wrong magic values that repeat in as short as 200 or fewer values! And, as anyone who has used RND knows, these short sequences happen often enough to be a serious problem.

Actually, it is super difficult to fake generation of "random" numbers. There is level upon level of subtlety in the math involved in proving that any system for generating pseudo-random numbers is in fact able to provide truly random results.

What we should be worried about is something useful enough to appear random, even if it might eventually fail some exotic randomness test. It turns out that there is a very simple and devastatingly powerful way to test for randomness. Just put random dots on the HIRES screen. If the screen turns white, you are well on your way to having a good random number generator. If it gets lines, large patterns, or shading in it during this test, you have preferential numbers. If the screen "sticks" and never gets to all white, your sequence is too short to be useful and is repeating itself.

This simple scheme uses your eye as an optical correlator to really pull any nonrandomness right out of the woodwork.

Applesloth's RND always fails the screen test. Sometimes it fails it quickly, "sticking" after as few as 200 dots. Other times, you will get thousands of dots on the screen before the sequence repeats. The worst results are gotten by rerunning the same sequence over and over again.

Want to try it? . . .

```
3   REM
             DEMO TO SHOW WHY RND AINT

4   REM
             * RUN IT TILL IT STICKS *

5   HGR : HCOLOR = 3: REM

10  X = 280 * RND (1): Y = 192 * RND
        (1): HPLOT X,Y: GOTO 10
```

There are a few [J]'s in and amongst the code in this listing for pretty printing. Leave them off if you care to. Unless you immediately happen into the short sequences, the program may have to run a few minutes before it sticks.

Actually, to be fair, Applesloth is stuck with doing floating point random number generation, which is a far stickier problem than simply generating one number from a small integer field.

An Integer Pseudo-Random Generator

Let's instead worry about generating integer pseudo-random numbers. The method we will show you easily handles any value from RND (2) to RND (255), and is extremely fast. It passes the screen-fill test with flying colors.

First, some theory. We will use a method called the *shift register pseudo-random sequence generator* method. This one is detailed both in the *TTL Cookbook* and the *CMOS Cookbook* (Sams 21035 and 21398).
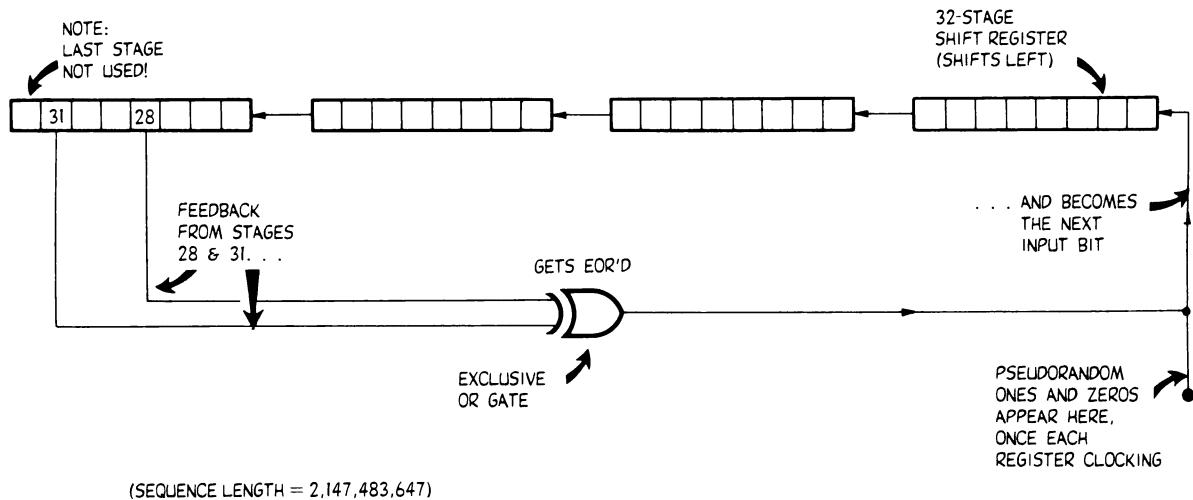
There is a hardware beastie called a shift register that can be made to behave like a counter. By taking certain high taps off the shift register and EXCLUSIVE-ORing them together and feeding these back to the input, you can generate a very long sequence.

Very handily, any tests you make on a short burst in the sequence will lead you to believe you have a true random number generator. The optimum feedback connections lead to a *maximal length* sequence, which turns out to be one less than two raised to the number of stages in use.

To get "random" numbers, you keep picking up new numbers in the sequence, or else jump to some other wildly different place in the series. To get replays or noise that repeats, you start over again at the same point in the series you did before.

We will use a 31-stage pseudo-random register since the feedback needed is simpler than that needed by a 32-stage one. The hardware we are going to synthesize with software looks like so . . .

A HARDWARE WAY TO GENERATE "RANDOM" NUMBERS:

(SEQUENCE LENGTH = 2,147,483,647)

There are 31 stages to our register. We take the output from stage 31 and EXCLUSIVE-OR it with the output from stage 28. The EOR of these taps then becomes the new value fed back to the input. These stage taps are "magic" values; anything else won't give you a super long series. We've shown this as a "shift-left" register, so we can be comparable to the replacement software we are about to use.

The sequence you get is one less than 2^{31}, which translates to 2,147,483,647 counts before repeating. The variable sequence length of the Applesloth code is avoided, since you have one and only *one* long sequence, rather than bunches, a few of which can end up short.

This shift register can be thought of as a bit pipe or stream with two billion marbles in it, half red and half white. Grab any four marbles in sequence and you have a 4-bit random number. Grab the next four and you have a new 4-bit "random" number, and so on. In this case, you can get half a billion different 4-bit random numbers in sequence before the same marbles start coming back out. And, in fact, you will get four different half billion number sequences that are predictably related but not the same, since you are one marble short at the end of the first run, and so on.

Bunches, at any rate.

There is only one little gotcha to using a generator like this. What about the missing count? It turns out that . . .

A GOTCHA—

A pseudo-random sequence generator will hang if it ever gets into the "all zeros" state.

DON'T LET THIS HAPPEN!

Now, the odds are only one in two billion of this ever happening, but you should know about it, and should prevent this hangup from ever happening. All you do is make sure there is a one somewhere in your shift register before you begin.

We will use software rather than hardware here. Set aside four bytes for the needed 31 bits. Use the EOR command for the EXCLUSIVE-OR logic, and use shift commands to move the bits from stage to stage.

Some Code

It takes more than just a pseudo-random generator to make a good random number generation system.

First, we should have some way of initializing or reseeding the PSR 4-byte shift register. We do this by grabbing two bytes from RNDL and RNDH that are truly random, and by grabbing two more bytes off the last PSR state.

Secondly, we need some way to get an old sequence back for replays and noise that repeats. To do this, we keep a copy of the old reseeding in a separate 4-byte seed register. For a new sequence, you load the PSR from the reseeder. For a "used" or repeat sequence, you reload the PSR from the seed register.

Thirdly, we need some way to deal with nonbinary numbers. A RND (32) is fairly trivial, since 32 is a binary number, and we expect a result anywhere between 0 and 31. To do this, just whump the PSR register five times, once for each bit, and read the bits with a $1F mask (that's 0001 1111 in binary) to get your result. For different binary lengths, use different mask lengths. The magic mask values are $01, $03, $07, $0F, $1F, $3F, $7F, and $FF.
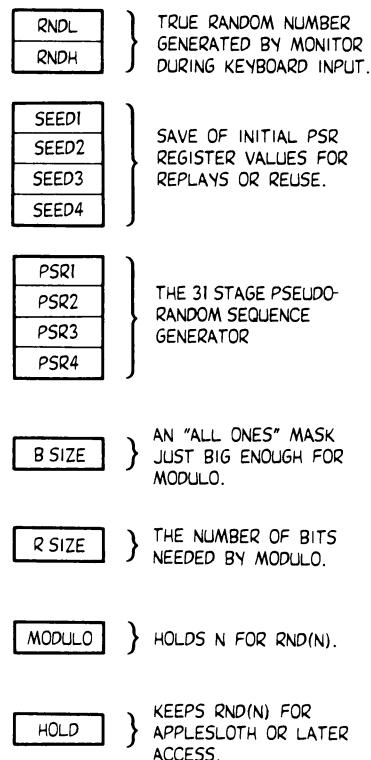
But what about a RND (26)?

Here we expect a result between decimal 0 and 25, or between hex $00 and $19. What you do is use a mask to grab more than enough bits off the PSR, and then compare the result. If the result is in range, use it. If not, go fish. Repeat the process as often as you have to.

Which I'm not very proud of, but it works. For nonbinary values, there will be some chance of having to repeat the process. This chance is always less than 50 percent worst case, and typically, is much better. So, you will still get a fast result on any RND choice although binary values will be the fastest.

Since there are lots of pieces to this randomizer, let's first look at our working stashes to see what they tell us . . .

STASHES USED BY RANDOM:

RNDL	⎫ TRUE RANDOM NUMBER
RNDH	⎬ GENERATED BY MONITOR ⎭ DURING KEYBOARD INPUT.

SEED1	⎫
SEED2	⎬ SAVE OF INITIAL PSR REGISTER VALUES FOR
SEED3	REPLAYS OR REUSE.
SEED4	⎭

PSR1	⎫
PSR2	⎬ THE 31 STAGE PSEUDO-RANDOM SEQUENCE
PSR3	GENERATOR
PSR4	⎭

B SIZE	⎬ AN "ALL ONES" MASK JUST BIG ENOUGH FOR MODULO.

R SIZE	⎬ THE NUMBER OF BITS NEEDED BY MODULO.

MODULO	⎬ HOLDS N FOR RND(N).

HOLD	⎬ KEEPS RND(N) FOR APPLESLOTH OR LATER ACCESS.

There are fourteen stash values involved.

The actual PSR generator is labeled PSR1 through PSR4. We input to the low bit of PSR1 and feedback from bits 28 and 31 that are stashed in PSR4.

There are four seed bytes used to hold the previous starting point for the PSR sequence. These are called SEED1 through SEED4. These locations are seeded from the monitor's RNDL and RNDH.

The location called MODULO holds your RND argument. For instance, on a single die, use a MODULO value of six. In return, you will get one of the six possible equiprobable states from zero to five back. MODULO must be set on first use, but if you want the same random range over and over again, you do not have to change it.

The locations called RSIZE and BSIZE take some explanation, since they are the key to generating nonbinary random values. BSIZE is a mask of enough ones to equal one less than the next higher binary power of the number you are after. That's one of those magic $01, $03, $07 . . . through $FF values. BSIZE is automatically calculated for you when and as needed. RSIZE is a save of the number of PSR advances needed to get enough bits to handle your MODULO.

For instance, on a die, BSIZE will be a $07, or binary %0000 0111, while RSIZE will be three. Why? Because it will take three bits to generate one of the numbers from zero to five. Should we overdo ourselves and get a six or seven result, we go fish and try again. The odds of hitting a legal value in this case are 3/4 of the time on the first try, and 15/16 of the time by the second try.

The reason you want to keep BSIZE as small as possible is so your odds of a hit are high. If, instead, you tried for six values out of a possible 256, your odds on a first-try hit will be a miniscule 6/256. The rea-

son for a separate save of RSIZE is so you do not have to recalculate BSIZE for each entry.

Which speeds things up bunches.

There are several places where our code *falls through* to another routine . . .

> **FALLING THROUGH—**
>
> **Code that automatically goes on and does a second task.**
>
> **You also have the option of doing only the second task by itself.**

There are three parts to the pseudo-random generator code. These are the reseeder, the N initializer, and the actual PSR generator.

Each part is simple enough that you should be able to work up your own flowcharts.

The reseeder is used to move your position in the PSR sequence either to where you last started counting, or else to some wildly new point.

You should always JSR to this code anytime you want to start randomizing something new. If you do a JSR RESEED, you will shuffle the deck and begin at some unknown point in the PSR sequence. If you do a JSR RESET, you will reload the last seed value you used. Use RESEED for something entirely new. Use RESET to repeat the last sequence of random numbers for a replay or for noise that repeats.

Note how RESEED falls through to RESET. Note also that we make sure that PSR2 is not a zero value. If it is, we force it to one. This is one heavy way to be sure that you never hit the all ones gotcha in your PSR generator.

Every time you start up, or every time you change your modulo, you will have to activate the N initializer. Do this by a JSR to RANDOM, after putting the number of possible values you are after into MOD-ULO. MODULO must be at least one. If it is zero, an error trap increments it.

Now, the code in the N initializer is admittedly obtuse, but this is what it does: Your modulo is scanned to generate a BSIZE mask with just enough sequential ones in it to equal or exceed your modulo value. At the same time, the number of bits involved is saved as RSIZE.

For instance, say you want RND (10). Modulo will be ten, and you expect the ten digits from zero to nine back. BSIZE will be %0000 1111, since this is the smallest mask you can have that can isolate all the digits from zero to nine. RSIZE will be four, since four random bits are needed from the PSR generator.

The N initializer falls through to the "real" PSR generator. You have to use this initializer any time you first begin or any time you change your modulo.

There are two parts to the "real" PSR generator. The first half, labeled REUSEN, gets enough bits to be equal to or more than your modulo.

To do this, REUSEN first "aligns" bit 28 to bit 31 of PSR4 using the accumulator to shift bit 28 over three places. An EOR then computes the feedback term. This particular EOR term gets shifted into the carry.

Note that there are seven worthless EOR bit results calculated at the same time. These are simply ignored. The good result ends up in the carry flag; everything else gets flushed.

Our carry flag now holds our feedback product. To shift our shift register, you move carry into the least significant bit of PSR1 and shift the rest of PSR1's bits one to the left. The high bit now goes into the carry. Now shift, in turn PSR2, PSR3, and PSR4. The net result is that you faked a pseudo-random shift register with software.

The PSR is shifted as often as you have to, once for each count of RSIZE. Three whaps for a die, four for modulo ten, and so on.

At this point, enough bits have been randomized in PSR1 to give you a totally new number equal to or larger than your modulo.

The second half of the PSR process is called RANGE. Its purpose is to see if you are within your modulo with the present random value in PSR1. If you are in range, you are finished. If not, you have to repeat the process as often as you need to for a useful result. As we've seen, the odds on a hit are always greater than 50 percent and are usually much greater.

RANGE grabs the value in PSR1 and masks it with BSIZE. This cuts the number down to size, such as to four bits for a modulo of ten. The same four bits could be used for any modulo from nine through sixteen.

The result in the accumulator is then compared to MODULO. If you are *less than* MODULO, then your value is acceptable. If you are *equal to or greater than* MODULO, then your present value is no good, and you need another trip back through the PSR.

Two minor points. Note that you do not have to go back through the N initializer for a repeat trip, since you already know and have saved BSIZE and RSIZE. This speeds things up considerably. Secondly, note that you always want less than MODULO as a result, because of a possible zero answer. To repeat, if your MODULO is six, you get any of the six different values of zero, one, two, three, four, or five. You do *not* want an answer of "six" for a MODULO of six, since this is the *seventh,* and not the sixth possible value.

Summing up, to generate a "random" number, put the range of that number into MODULO. If this is your first time through, do a JSR RESEED. If you want to repeat a previous series, do a JSR RESET. Then do a JSR RANDOM. Your result ends up in the accumulator for machine language use, and in HOLD either for high level language access or for future reference.

If you want to use the same modulo over again, do a JSR REUSEN. This is much faster.

The companion demo is called FILL. It fills the HIRES screen the same way that WHY RND AIN'T didn't. Note particularly the speed difference, how clean the process is, and how you eventually get to a totally white screen.

Even this speed test is hardly fair, since we are still using the ludicrously slow Applesloth HPLOT subroutines in the demo. You can go much faster if you add your own custom HPLOT code.

You can extend your modulo to 511 by making two trips to RANDOM. To do this, divide the *even* part of your modulo *by two.* Generate this value and double it. Then do a separate RND (2) to pick even or odd results.

MIND BENDERS

— Show how to eliminate the repeat trips for nonbinary N.

— What is the actual speed involved in generating a random number?

— How can you use a PSR generator to generate speaker noise?

— Why and how does a 23-bit PSR fail the screen-fill test?

— Can you think of any uses for shorter or longer PSR generators?

```
PROGRAM RM-7
      RANDOM NUMBERS
```

```
----- NEXT OBJECT FILE NAME IS RANDOM
6F00:             3            ORG   $6F00      ; PUT MODULE #7 AT $6F00


6F00:             5 ;  ******************************************
6F00:             6 ;  *                                        *
6F00:             7 ;  *               -< RANDOM >-             *
6F00:             8 ;  *                                        *
6F00:             9 ;  *     (PSEUDORANDOM INTEGER GENERATOR)    *
6F00:            10 ;  *                                        *
6F00:            11 ;  *       VERSION 1.0   ($6F00-$6FB2)       *
6F00:            12 ;  *                                        *
6F00:            13 ;  *                 1-12-83                 *
6F00:            14 ;  *........................................*
6F00:            15 ;  *                                        *
6F00:            16 ;  *           COPYRIGHT C 1983 BY           *
6F00:            17 ;  *                                        *
6F00:            18 ;  *     DON LANCASTER AND SYNERGETICS       *
6F00:            19 ;  *     BOX 1300, THATCHER AZ., 85552       *
6F00:            20 ;  *                                        *
6F00:            21 ;  *     ALL COMMERCIAL RIGHTS RESERVED      *
6F00:            22 ;  *                                        *
6F00:            23 ;  ******************************************


6F00:            25 ;            *** WHAT IT DOES ***

6F00:            27 ;  THIS MODULE GIVES YOU A PSEUDORANDOM INTEGER FROM A
6F00:            28 ;  FIELD OF N.  N CAN RANGE FROM 2 TO 255.
6F00:            29 ;


6F00:            31 ;            *** HOW TO USE IT ***

6F00:            33 ;  TO RESEED (INITIALIZE) FROM A TRUE RANDOM NUMBER,
6F00:            34 ;  DO A JSR SEED WITH A JSR $6F2E OR A CALL 28462.
6F00:            35 ;
6F00:            36 ;  TO REPEAT AN OLD PSEUDORANDOM SERIES, DO A JSR RESET
6F00:            37 ;  BY DOING A JSR $6F44 OR A CALL 28484.
6F00:            38 ;
6F00:            39 ;  TO GET A PSEUDORANDOM VALUE:
6F00:            40 ;
6F00:            41 ;     FROM MACHINE LANGUAGE, PUT N IN THE ACCUMULATOR
6F00:            42 ;     AND THEN JSR RANDOM AT $6F5B.  RND(N) RETURNS IN A.
6F00:            43 ;
6F00:            44 ;     FROM APPLESOFT, STORE N IN MODULO AT 28593
6F00:            45 ;     AND THEN CALL RANDHL AT 28504.  RND(N) ENDS
6F00:            46 ;     UP IN HOLD AT 28594.
```

PROGRAM RM-7, CONT'D . . .

```
6F00:            49 ;              *** GOTCHAS ***

6F00:            51 ;    HALF THE ORIGINAL RANDOM SEED COMES FROM RNDL AND
6F00:            52 ;    RNDH IN THE MONITOR.  THE OTHER HALF COMES FROM
6F00:            53 ;    THE PREVIOUS PSR SEQUENCE.
6F00:            54 ;    N VALUES ONE LESS THAN A BINARY POWER EXECUTE FASTEST.
6F00:            55 ;    APPLESOFT IS NEEDED FOR THE SCREENFILL DEMO.
6F00:            56 ;    THE A AND Y REGISTERS ARE USED BY THESE SUBS.


6F00:            58 ;              *** ENHANCEMENTS ***

6F00:            60 ;    THE DEMO "FILL" LETS YOU FILL THE HIRES SCREEN RANDOMLY.
6F00:            61 ;    RUN IT WITH A JSR $7E00 OR A CALL 332256.
6F00:            62 ;


6F00:            64 ;              *** RANDOM COMMENTS ***

6F00:            66 ;    VALUES OF N THAT ARE NOT EQUAL TO ONE LESS THAN A
6F00:            67 ;    POWER OF TWO MAY NEED REPEAT TRIPS THROUGH THE PSR
6F00:            68 ;    SEQUENCER.  THIS IS DONE AUTOMATICALLY.  THE PROBABILITY
6F00:            69 ;    OF A HIT ALWAYS EXCEEDS 50% WORST CASE PER PASS AND
6F00:            70 ;    IS USUALLY MUCH HIGHER.
6F00:            71 ;
```

PROGRAM RM-7, CONT'D . . .

```
6F00:              74 ;              *** HOOKS ***


F3E2:              76 HGR     EQU  $F3E2      ; APPLESOFT CLEAR TO HIRES ONE
F457:              77 HPLOT   EQU  $F457      ; APPLESOFT HIRES PLOT
C000:              78 IOADR   EQU  $C000      ; KEYBOARD
C010:              79 KBSTR   EQU  $C010      ; KEYBOARD RESET
004E:              80 RNDL    EQU  $4E        ; RANDOM NUMBER LOW
004F:              81 RNDH    EQU  $4F        ; RANDOM NUMBER HIGH
F6EC:              82 SETHCOL EQU  $F6EC      ; APPLESOFT HIRES COLOR SET
C050:              83 TEXT    EQU  $C050      ; TEXT SCREEN




6F00:              85 ;              *** CONSTANTS ***


0003:              87 COLOR   EQU  $03        ; FOR A WHITE PLOT
```

PROGRAM RM-7, CONT'D . . .

```
6F00:           90 ;                *** SCREENFLL DEMO ***


6F00:           92 ;        THIS DEMO FILLS THE HIRES SCREEN ONE RANDOM
6F00:           93 ;        DOT AT A TIME.
6F00:           94 ;
6F00:           95 ;
6F00:           96 ;
6F00:           97 ;


6F00:20 E2 F3   99 FILL     JSR  HGR        ; CLEAR HIRES SCREEN
6F03:A2 03     100          LDX  #COLOR     ; PICK COLOR (03=WHITE)
6F05:20 EC F6  101          JSR  SETHCOL    ;


6F08:20 2E 6F  103          JSR  RESEED     ; SEED PSR FROM RNDL,RNDH

6F0B:A9 BF     105 PLOTDOT  LDA  #$BF       ; 191 DOTS HIGH
6F0D:8D B1 6F  106          STA  MODULO     ;
6F10:20 5B 6F  107          JSR  RANDOM     ; GET RANDOM H
6F13:48        108          PHA             ; AND SAVE ON STACK
6F14:A9 FF     109          LDA  #$FF       ; 256 DOTS WIDE
6F16:8D B1 6F  110          STA  MODULO     ;
6F19:20 5B 6F  111          JSR  RANDOM     ; GET VERT
6F1C:A0 00     112          LDY  #$00       ; NO HISCREEN
6F1E:AA        113          TAX             ; TRANSFER H
6F1F:68        114          PLA             ; GET V
6F20:20 57 F4  115          JSR  HPLOT      ; PLOT DOT ON SCREEN
6F23:2C 00 C0  116          BIT  IOADR      ; READ KEYBOARD
6F26:30 02     117          BMI  EXIT7      ;
6F28:10 E1     118          BPL  PLOTDOT    ; CONTINUE IF NO KP
6F2A:2C 10 C0  119 EXIT7    BIT  KBSTR      ; RESET KEYBOARD
6F2D:60        120          RTS             ; AND QUIT
```

PROGRAM RM-7, CONT'D . . .

```
6F2E:           123 ;              ***   PSEUDORANDOM GENERATOR ***

6F2E:           125 ;   THE PSEUDORANDOM GENERATOR IS A REGISTER THAT IS 31
6F2E:           126 ;   BITS LONG.  BITS 28 AND 31 ARE EXCLUSIVE ORED TO SET
6F2E:           127 ;   THE NEXT MSB.  SEQUENCE LENGTH IS 2,147,483,647.
6F2E:           128 ;
6F2E:           129 ;
6F2E:           130 ;
6F2E:           131 ;


6F2E:           133 ;              *** THE RESEEDER ***


6F2E:A5 4E      135 RESEED  LDA   RNDL       ; GET RANDOM NUMBER
6F30:8D A7 6F   136         STA   SEED1      ; FROM MONITOR KEYBOARD RND
6F33:A5 4F      137         LDA   RNDH       ; AND STORE FOR PSR SEED.
6F35:8D AA 6F   138         STA   SEED4      ;
6F38:AD AD 6F   139         LDA   PSR3       ; RESEED MIDDLE FROM OLD
6F3B:8D A8 6F   140         STA   SEED2      ;
6F3E:AD AC 6F   141         LDA   PSR2       ;
6F41:8D A9 6F   142         STA   SEED3      ; AND FALL THRU TO RESET


6F44:A0 04      144 RESET   LDY   #$04       ; MOVE SEED TO PSR REGISTER
6F46:B9 A7 6F   145 NXT7    LDA   SEED1,Y    ;
6F49:99 AB 6F   146         STA   PSR1,Y     ;
6F4C:88         147         DEY              ;
6F4D:D0 F7      148         BNE   NXT7       ;

6F4F:AD AC 6F   150         LDA   PSR2       ; FORCE PSR SEED TO NONZERO
6F52:D0 03      151         BNE   DONE7      ; BY FORCING NONZERO PSR2
6F54:EE AC 6F   152         INC   PSR2       ;
6F57:60         153 DONE7   RTS              ; AND RETURN
```

PROGRAM RM-7, CONT'D . . .

```
6F58:              156 ;        *** THE N INITIALIZER ***


6F58:AD B1 6F      158 RNDHL   LDA  MODULO   ; ENTER HERE FROM APPLESOFT
6F5B:9D B1 6F      159 RANDOM  STA  MODULO   ; ENTER HERE FROM MACHINE LANGUAGE
6F5E:D0 05         160         BNE  BSCALC   ; N MUST NOT BE ZERO!
6F60:A9 02         161         LDA  #$02     ; USE N=2 MINIMUM
6F62:8D B1 6F      162         STA  MODULO   ;

6F65:A9 FF         164 BSCALC  LDA  #$FF     ; INIT SIZE TO 255
6F67:8D AF 6F      165         STA  BSIZE    ; ENOUGH ONES HERE > MODULO
6F6A:A0 08         166         LDY  #$08     ; FOR 8 BITS
6F6C:AD B1 6F      167         LDA  MODULO   ; GET MODULO AND CALCULATE
6F6F:2A            168 SMALLER ROL  A        ; NEXT LARGER
6F70:B0 0C         169         BCS  ADVANCE  ;
6F72:4E AF 6F      170         LSR  BSIZE    ; DIVIDE BY TWO
6F75:88            171         DEY           ; NEXT SMALLER
6F76:D0 F7         172         BNE  SMALLER  ;
6F78:8C B0 6F      173         STY  RSIZE    ; SAVE FOR RETRY



6F7B:              176 ;        *** THE ACTUAL PSR GENERATOR ***


6F7B:AC B0 6F      178 REUSEN  LDY  RSIZE    ; RESTORE IF RETRY
6F7E:AD AE 6F      179 ADVANCE LDA  PSR4     ; GET HIGH PSR
6F81:0A            180         ASL  A        ; ALIGN BIT 28 TO 31
6F82:0A            181         ASL  A        ;
6F83:0A            182         ASL  A        ;
6F84:4D AE 6F      183         EOR  PSR4     ; AND EXCLUSIVE OR
6F87:0A            184         ASL  A        ; MOVE TO CARRY
6F88:0A            185         ASL  A        ;
6F89:2E AB 6F      186         ROL  PSR1     ; SHIFT LOW PSR
6F8C:2E AC 6F      187         ROL  PSR2     ; SHIFT NEXT PSR
6F8F:2E AD 6F      188         ROL  PSR3     ; AND ONCE MORE
6F92:2E AE 6F      189         ROL  PSR4     ; FINALLY THE HIGH BYTE
6F95:88            190         DEY           ; REPEAT FOR EVERY BIT IN BSIZE
6F96:D0 E6         191         BNE  ADVANCE  ;


6F98:AD AB 6F      193 RANGE   LDA  PSR1     ; GET VALUE
6F9B:2D AF 6F      194         AND  BSIZE    ; MASK NEXT BINARY VALUE
6F9E:CD B1 6F      195         CMP  MODULO   ; IS VALUE TOO BIG?
6FA1:B0 D8         196         BCS  REUSEN   ; YES, TRY AGAIN
6FA3:8D B2 6F      197         STA  HOLD     ; SAVE VALID PSR
6FA6:60            198         RTS           ; AND EXIT
```

PROGRAM RM-7, CONT'D . . .

```
6FA7:            201 ;              *** PSR REGISTERS ***

6FA7:            203 ;     SEEDL AND SEEDH HOLD THE STARTING SEED SHOULD YOU
6FA7:            204 ;     WANT TO RERUN THE SERIES.  PSR1, PSR2, PSR3, AND
6FA7:            205 ;     PSR4 FORM THE 23 BIT PSEUDORANDOM SEQUENCER.
6FA7:            206 ;
6FA7:            207 ;     BSIZE IS A SIZING MASK.
6FA7:            208 ;
6FA7:            209 ;     MODULO HOLDS THE VALUE N, WHILE HOLD KEEPS THE RANDOM (N)
6FA7:            210 ;


6FA7:AA          212 SEED1    DFB    $AA       ; SEED LOW VALUE
6FA8:AA          213 SEED2    DFB    $AA       ; SEED SECOND LOWEST
6FA9:AA          214 SEED3    DFB    $AA       ; SEED THIRD LOWEST
6FAA:AA          215 SEED4    DFB    $AA       ; HIGH SEED
6FAB:AA          216 PSR1     DFB    $AA       ; PSR LOW BYTE
6FAC:AA          217 PSR2     DFB    $AA       ; PSR SECOND LOWEST
6FAD:3B          218 PSR3     DFB    $3B       ; PSR THIRD LOWEST
6FAE:AA          219 PSR4     DFB    $AA       ; PSR HIGHEST
6FAF:FF          220 BSIZE    DFB    $FF       ; SAVE OF BINARY SIZE
6FB0:04          221 RSIZE    DFB    $04       ; YSAVE FOR RETRY

6FB1:07          223 MODULO   DFB    $07       ; MAXIMUM SIZE OF N
6FB2:00          224 HOLD     DFB    $00       ; SAVE OF PSR VALUE
```

SHUFFLE

a fast "random exchange"
method of rearranging cards or
number arrays

There are lots of computer situations where you might like to take a pile of objects and rearrange them into some different order.

Shuffling a deck of cards is the most obvious example of this sort of thing. You might use playing cards for poker or blackjack simulations. Other times, the cards may have different symbols or messages on them. Tarot cards are an example, as are the Chance and Community Chest decks in a Monopoly simulation.

The things being shuffled need not be paper cards, of course. They could be tiles in a magic number game, letters in a word, the sequence in which new things appear, a maze in an adventure, or a journey into cryptography.

The fancy name for shuffling is randomizing without replacement. In randomizing without replacement, you simply rearrange a fixed array of values that you already have on hand. Once drawn from the deck, the four of clubs will not reappear.

The random number generator of the last ripoff module kept all the marbles in the pipe. You just cloned off the marbles you wanted. This was randomizing with replacement. In randomizing with replacement, the same value can come up over and over again.

Hence . . .

Randomizing WITH Replacement—

Grabbing a random number without removing that number from being available for future grabs.

Rolling a die is an example.

Randomizing WITHOUT Replacement—

Grabbing a random number while also eliminating the availability of that number for future grabs.

Shuffling cards is typical.

Note that these are totally different things. You'll get absurd results if you try to use the wrong one. Like only six different throws of a die before the die is "empty." Or the nine of spades dealt to you three cards in a row.

To throw some other terms at you, grabbing without replacement involves an *infinite pool* of numbers. Or at least an irrigation ditch full.

Grabbing with replacement involves a *finite pool* of numbers. These numbers are usually arranged into a fixed and rather small *array*. The array size on a playing card deck is usually 52.

The typical way that beginners try to shuffle things on their Apple has two very serious flaws. First, of course, they will be trying to use the Applesoft RND subroutine, which, as we have seen, is not.

Besides being rather slow.

We can easily fix this particular hassle by switching to the random number generator of the last ripoff module.

The second problem is more subtle. If you grab 52 random numbers in a row, you have to check each *new* number to make sure it was not duplicated before. This is no problem on the first card, and is trivial on the first few cards. But on, say card 50, the odds are 50/52 that you already have this card and have to go back again and again.

In fact, for your last card, you might need 52 *additional* tries to pick up only a 0.63 odds of finding the remaining card.

1/e and all that statistical stuff.

You, in fact, have to deal hundreds or even thousands of cards to be reasonably sure of getting 52 different ones. So, testing for duplicates is a bad scene because it takes ridiculously long and involves many wasted trips to the random number generator.

Let's work smarter and not harder. Do not try to take your numbers out of an infinite pool. Instead, take them out of a small and fixed array. Center your activities on rearranging the array.

Here is a good and fast way to shuffle a pile of something . . .

TO REARRANGE N OBJECTS—

Take the object in the first location and interchange it with an object in another location in the array, chosen at random.

Then take the object in the second location and do the same thing.

Repeat this for all the locations.

In other words, lay your 52 cards on the table. Grab the first card and interchange it with any card, picked at random. Next, grab the second card and interchange it with any card, again picked at random. Continue the process till you run out of cards.

Note two things. First, there are only 52 random numbers needed this way, since each random number gets used only once. Secondly, a card in some position will sometimes replace itself. This happens if the card in location number seven is interchanged with the random location number seven that just came up.

The odds on a card replacing itself are exactly the same as shuffling a real card deck and having the same card end up in the same position.

Which is rare but it certainly can happen. You can even get the deck back exactly the way you started. Odds on this are a tad low, though. The key point is that this random interchange method exactly duplicates a fair and thorough shuffle of real cards.

The same thing works for other shuffles. For a 15-tile magic square, you only interchange 15 values. You only swap six letters to jumble a six letter word, and so on.

Let's try it.

A Shuffler

The subroutine called SHUFFLR will take an array named CARDECK of length ARNUM and reorder everything.

SHUFFLR does this by using the random number generator of the previous ripoff module. CARDECK is presently set up to hold 52 cards, and ARNUM equals decimal 52 or hex $34.

The shuffling process is done by taking the first array value and interchanging it with an array value in a slot chosen at random. To find the exchange slot, you get a random number from 0 to 51 and do the exchange. The process gets repeated 52 times, thus swapping each card with some other card or itself at least once.

One of the array values being swapped is temporarily stashed on the stack. This handles the juggling process of moving two things between two locations without dropping either one of them.

To use SHUFFLR for other tasks, you just change the array values and the size of your array.

Note that SHUFFLR does not care what is inside each array slot. This lets you use meaningful codes for each array value. These codes are totally independent of the shuffling process.

How do you code a deck of cards?

One way to code the cards is to use one hex digit for the values

from ace through king. An obvious choice is to use $X1 for an ace, $X2 for a two, $X9 for a nine, $XA for a ten, $XB for a jack, and so on. Let's use the least significant hex digit for this.

We will use the other hex digit to pick a suit. Say $0X for hearts, $1X for diamonds, $2X for clubs, and $3X for spades. Thus, the ace of spades will be coded $31, while the king of diamonds will be a $1D.

Clear?

A companion demo called DEALER will exercise your shuffler. The "S" key will shuffle the deck. The "R" key will repeat the previous shuffle for a replay. The "D" key, or optionally, the spacebar deals a card. The "Q" key quits the program for you.

We have used the short file printing method to handle text. It is the better choice here because we have lots of short and ordered words that we need in a more or less random access way. We are also under the nasty 256 character limit here. Use of a short file also saves dragging IMPRINT into the demo. It also gives you a chance to play with absolute indexed addressing.

The four options needed are simple enough that we will handle them by brute force, rather than using fancier option picker code.

We have also included a card counter. This one can be used for a position or a score, and does not need any hex-to-decimal or decimal-to-hex conversion. I'll leave it to you to puzzle out how this one works.

Naturally, a machine language randomizing-without-replacement module is far too fast to use as a real time playing card shuffle. So, we'll have to slow it down bunches. To do this, we will build the shuffler into a sound effect that mimics a deck being shuffled, and adjust the timing to "real" time.

Should you need something rearranged very quickly, be sure to defeat the sound effects. Or else adjust the effects to mimic what you are emulating.

Once again, the pseudo-random generator of the previous ripoff module is needed to get this module to work. So, be sure to have either RANDOM or THE WHOLE BALL OF WAX in your machine when using either the shuffler or the card demo.

MIND BENDERS

—What is the total time needed to shuffle a deck of 52 cards, with and without the sound effects?

—Add a HIRES or LORES graphics display of the playing cards.

—Show how to do an "instant solver" for those word jumble puzzles on a newspaper's comics page.

—In a word guessing game that has a file of hundreds of words, show how to get each word just once yet in a different order each session.

—Why does the card number display work in decimal without needing any hex conversion?

—What changes are needed to make the demo professionally useful?

PROGRAM RM-8
SHUFFLE

```
----- NEXT OBJECT FILE NAME IS SHUFFLER
7000:             3          ORG  $7000      ; PUT MODULE #8 AT $7000


7000:             5 ;   *****************************************
7000:             6 ;   *                                       *
7000:             7 ;   *            -< SHUFFLER >-             *
7000:             8 ;   *                                       *
7000:             9 ;   *   ( RANDOMIZING WITHOUT REPLACEMENT )  *
7000:            10 ;   *                                       *
7000:            11 ;   *      VERSION 1.0   ($7000-$7246)       *
7000:            12 ;   *                                       *
7000:            13 ;   *             5-24-83                    *
7000:            14 ;   *.....................................*
7000:            15 ;   *                                       *
7000:            16 ;   *          COPYRIGHT C 1983 BY          *
7000:            17 ;   *                                       *
7000:            18 ;   *     DON LANCASTER AND SYNERGETICS     *
7000:            19 ;   *     BOX 1300, THATCHER AZ., 85552     *
7000:            20 ;   *                                       *
7000:            21 ;   *     ALL COMMERCIAL RIGHTS RESERVED    *
7000:            22 ;   *                                       *
7000:            23 ;   *****************************************


7000:            25 ;            *** WHAT IT DOES ***

7000:            27 ;   THIS MODULE SHOWS YOU HOW TO SHUFFLE OR REARRANGE
7000:            28 ;   AN ARRAY OF CARDS, NUMBERS, LETTERS, OR OBJECTS.
7000:            29 ;


7000:            31 ;            *** HOW TO USE IT ***

7000:            33 ;   TO USE THE SHUFFLER:
7000:            34 ;
7000:            35 ;       START YOUR ARRAY FILE WITH CARDECK AT $7213.
7000:            36 ;       PUT THE NUMBER OF ARRAY ELEMENTS IN ARNUM AT $710E.
7000:            37 ;       THEN JSR SHUFFLR AT $70F1.  EQUIVALENT APPLESLOTH
7000:            38 ;       LOCATIONS ARE 29203, 28942, AND 28913.


7000:            40 ;   TO RUN THE CARD DEALER DEMO:
7000:            41 ;
7000:            42 ;       JSR DEALER AT $7000 OR CALL 28672.
```

PROGRAM RM-8, CONT'D . . .

```
7000:          45 ;              *** GOTCHAS ***

7000:          47 ;    THE RANDOM SUBROUTINE MUST BE PRESENT IN THE
7000:          48 ;    MACHINE.  PRELOAD "RANDOM" OR "THE WHOLE BALL
7000:          49 ;    OF WAX" TO DO THIS.
7000:          50 ;
7000:          51 ;    YOUR ARRAY FILE MUST BE PRELOADED WITH THE
7000:          52 ;    PROPER VALUES.


7000:          54 ;            *** ENHANCEMENTS ***

7000:          56 ;    WORDS, OBJECTS, OR OTHER TYPES OF CARDS ARE DONE
7000:          57 ;    BY CHANGING THE MEANING AND SIZE OF YOUR ARRAY.
7000:          58 ;


7000:          60 ;            *** RANDOM COMMENTS ***

7000:          62 ;    THIS SHUFFLE DEMO IS INTENDED TO SHOW THE PROCESS
7000:          63 ;    INVOLVED.  AN ACTUAL CARD PROGRAM HAS TO BE FAR
7000:          64 ;    FRIENDLIER THAN THIS, AND SHOULD DISPLAY REAL CARDS.
7000:          65 ;
7000:          66 ;    THE DEMO ALSO SHOWS HOW TO HANDLE SIMPLE SCORING
7000:          67 ;    WITHOUT NEEDING HEX TO DECIMAL CONVERSION.
```

PROGRAM RM-8, CONT'D . . .

```
7000:              70 ;                  *** HOOKS ***


FDF0:              72 COUT1   EQU  $FDF0   ; OUTPUT TEXT TO SCREEN
FC58:              73 HOME    EQU  $FC58   ; CLEAR TEXT SCREEN AND HOME CURSOR
FB2F:              74 INIT    EQU  $FB2F   ; INITIALIZE TEXT SCREEN
C000:              75 IOADR   EQU  $C000   ; KEYBOARD INPUT LOCATION
C010:              76 KBDSTRB EQU  $C010   ; KEYBOARD STROBE RESET
FD1B:              77 KEYIN   EQU  $FD1B   ; MONITOR READKEY SUBROUTINE
FE80:              78 SETINV  EQU  $FE80   ; SET INVERSE SCREEN
FE84:              79 SETNORM EQU  $FE84   ; SET NORMAL SCREEN
C030:              80 SPKR    EQU  $C030   ; SPEAKER CLICK OUTPUT
FCA8:              81 WAIT    EQU  $FCA8   ; MONITOR TIME DELAY


6F5B:              83 RANDOM  EQU  $6F5B   ; RANDOM NUMBER INITIALIZER
6F2E:              84 RESEED  EQU  $6F2E   ; RANDOM NUMBER SEEDER
6F7B:              85 REUSEN  EQU  $6F7B   ; RANDOM NUMBER GENERATOR


0020:              87 WNDLFT  EQU  $20     ; LEFT SIDE OF SCROLL WINDOW
0021:              88 WNDWTH  EQU  $21     ; WIDTH OF SCROLL WINDOW
0022:              89 WNDTOP  EQU  $22     ; TOP OF SCROLL WINDOW
0023:              90 WNDBTM  EQU  $23     ; BOTTOM OF SCROLL WINDOW
0024:              91 CH      EQU  $24     ; CURSOR HORIZONTAL POSITION
0033:              92 PROMPT  EQU  $33     ; PROMPT SYMBOL


7000:              94 ;        *** TEXTFILE COMMANDS ***


0088:              96 B       EQU  $88     ; BACKSPACE
008D:              97 C       EQU  $8D     ; CARRIAGE RETURN
0084:              98 D       EQU  $84     ; DOS ATTENTION
009B:              99 E       EQU  $9B     ; ESCAPE
008A:             100 L       EQU  $8A     ; LINEFEED
0060:             101 P       EQU  $60     ; FLASHING PROMPT
0000:             102 X       EQU  $00     ; END OF MESSAGE
```

PROGRAM RM-8, CONT'D . . .

```
7000:              105 ;       *** DEALIN DEMO ***


7000:              107 ;     THIS DEMO EXCERCISES THE SHUFFLER
7000:              108 ;     ON A STANDARD DECK OF 52 CARDS.



7000:20 2F FB      110 DEALER  JSR   INIT      ; SET UP TEXT SCREEN
7003:20 58 FC      111         JSR   HOME      ; AND CLEAR IT
7006:20 2E 6F      112         JSR   RESEED    ; RESEED RANDOM
7009:AD 0E 71      113         LDA   ARNUM     ; GET ARRAY NUMBER
700C:20 5B 6F      114         JSR   RANDOM    ; INIT RANDOM


700F:A9 07         116         LDA   #$07      ; TAB 7 TO RIGHT
7011:85 24         117         STA   CH        ;
7013:20 80 FE      118         JSR   SETINV    ; INVERSE TITLE
7016:A0 5E         119         LDY   #>MS0-CV1 ; GET HEADER
7018:20 E3 70      120         JSR   TEXT8     ; AND DISPLAY
701B:20 84 FE      121         JSR   SETNORM   ; NORMAL TEXT
701E:A0 74         122         LDY   #>MS1-CV1 ; GET SCREEN PROMPTS
7020:20 E3 70      123         JSR   TEXT8     ; AND DISPLAY

7023:A9 07         125         LDA   #$07      ; SET LOWSCREEN WINDOW
7025:85 22         126         STA   WNDTOP    ;
7027:A9 05         127         LDA   #$05      ;
7029:85 20         128         STA   WNDLFT    ; TAB OVER TO CENTER
702B:A9 22         129         LDA   #$22      ;
702D:85 21         130         STA   WNDWTH    ;
702F:20 58 FC      131         JSR   HOME      ; GET INSIDE WINDOW

7032:2C 10 C0      133 CMND8   BIT   KBDSTRB   ; RESET KEYBOARD
7035:AD 00 C0      134 LOOK8   LDA   IOADR     ; READ KEYBOARD
7038:10 FB         135         BPL   LOOK8     ;
703A:2C 10 C0      136         BIT   KBDSTRB   ;
703D:C9 E1         137         CMP   #$E1      ; FORCE CASE
703F:90 02         138         BCC   CSORT     ;
7041:E9 20         139         SBC   #$20      ; SUBTRACT TO CHANGE CASE

7043:C9 D3         141 CSORT   CMP   #$D3      ; S FOR SHUFFLE?
7045:F0 1D         142         BEQ   SHUFF     ; YES
7047:C9 C4         143         CMP   #$C4      ; D FOR DEAL ?
7049:F0 2E         144         BEQ   DEAL      ;
704B:C9 A0         145         CMP   #$A0      ; ALSO SPACE FOR DEAL
704D:F0 2A         146         BEQ   DEAL      ;
704F:C9 D2         147         CMP   #$D2      ; R FOR REPLAY?
7051:F0 17         148         BEQ   REPLAY    ;
7053:C9 D1         149         CMP   #$D1      ; Q FOR QUIT?
7055:F0 06         150         BEQ   QUIT8     ;
7057:20 18 71      151         JSR   EFFECT2   ; BLORK
705A:4C 32 70      152         JMP   CMND8     ; TRY AGAIN FOR LEGAL KEY
```

PROGRAM RM-8, CONT'D . . .

```
705D:              155 ;        *** QUIT EXIT ***

7C5D:20 2F FB      157 QUIT8    JSR    INIT      ; OPEN WINDOW
7060:20 58 FC      158          JSR    HOME      ; CLEAR SCREEN
7063:60            159          RTS              ; AND EXIT ON "Q"


7064:              161 ;        *** SHUFF PROCESSING ***


7064:              163 ;        THIS CODE SHUFFLES THE DECK AND
7064:              164 ;        RESETS THE CARD COUNTERS TO ONE.


7064:20 F1 70      166 SHUFF    JSR    SHUFFLR   ; SHUFFLE THE DECK
7067:4C 6A 70      167          JMP    REPLAY    ; RESET COUNTERS




706A:              169 ;        *** REPLAY MODULE ***

706A:              171 ;        RESETS THE CARD COUNTER TO ZERO.


706A:A0 00         173 REPLAY   LDY    #$00      ; RESET COUNTERS
706C:8C EF 70      174          STY    HEXCNT    ;
706F:C8           175          INY              ; ONE MORE FOR PEOPLE
7070:8C F0 70      176          STY    DECCNT    ;
7073:20 58 FC      177          JSR    HOME      ; CLEAR OLD CARDS
7076:4C 32 70      178          JMP    CMND8     ; GO GET NEXT COMMAND
```

PROGRAM RM-8, CONT'D . . .

```
7079:             181 ;       *** DEAL PROCESSING ***


7079:             183 ;    THIS CODE TRYS TO DEAL A CARD IF
7079:             184 ;    THERE ARE ANY LEFT IN THE DECK.

7079:AD EF 70     186 DEAL    LDA   HEXCNT     ; GET NUMBER IN DECK (52)
707C:CD 0E 71     187         CMP   ARNUM      ; ANY CARDS LEFT?
707F:B0 5A        188         BCS   EMPTY8     ; NO, SAY SO

7081:A0 A5        190         LDY   #>MS2-CV1  ; SAY "CARD"
7083:20 E3 70     191         JSR   TEXT8      ;
7086:AD F0 70     192         LDA   DECCNT     ; GET TENS FOR CARD NUMBER
7089:4A           193         LSR   A          ;  AND SHIFT FOUR TO RIGHT
708A:4A           194         LSR   A          ;
708B:4A           195         LSR   A          ;
708C:4A           196         LSR   A          ;
708D:F0 05        197         BEQ   LOWDEC     ; IS IT NONZERO?
708F:09 B0        198         ORA   #$B0       ; CHANGE TO ASCII
7091:20 F0 FD     199         JSR   COUT1      ; AND PRINT IT
7094:AD F0 70     200 LOWDEC  LDA   DECCNT     ; GET UNITS FOR CARD NUMBER
7097:29 0F        201         AND   #$0F       ; MASK TENS
7099:09 B0        202         ORA   #$B0       ; CHANGE TO ASCII
709B:20 F0 FD     203         JSR   COUT1      ; AND PRINT IT

709E:A0 AB        205         LDY   #>MS3-CV1  ; SAY "IS THE"
70A0:20 E3 70     206         JSR   TEXT8      ;  TO SCREEN

70A3:AE EF 70     208         LDX   HEXCNT     ; GET CARD
70A6:BD 13 72     209         LDA   CARDECK,X  ; FROM DECK
70A9:48           210         PHA              ; AND SAVE FOR SUIT
70AA:29 0F        211         AND   #$0F       ; MASK SUIT
70AC:AA           212         TAX              ; USE AS INDEX
70AD:CA           213         DEX              ; MAKE ACE=1, NOT ZERO!
70AE:BC 2A 71     214         LDY   CARVAL,X   ; GET SUIT NAME
70B1:20 E3 70     215         JSR   TEXT8      ;  AND PRINT TO SCREEN

70B4:A0 B4        217         LDY   #>MS4-CV1  ; SAY "OF"
70B6:20 E3 70     218         JSR   TEXT8      ;   AND PRINT IT

70B9:68           220         PLA              ; GET CARD BACK
70BA:4A           221         LSR   A          ; AND SHIFT TO RIGHT
70BB:4A           222         LSR   A          ;
70BC:4A           223         LSR   A          ;
70BD:4A           224         LSR   A          ;
70BE:AA           225         TAX              ; USE AS INDEX
70BF:BC 37 71     226         LDY   CARSUIT,X  ; GET SUIT NAME
70C2:20 E3 70     227         JSR   TEXT8      ; AND PRINT IT
```

PROGRAM RM-8, CONT'D . . .

```
70C5:A0 B9      230           LDY  #>MS5-CV1 ; GET PERIOD AND CR
70C7:20 E3 70   231           JSR  TEXT8     ; AND PRINT IT
70CA:EE EF 70   232           INC  HEXCNT    ; GO TO NEXT CARD
70CD:F8         233           SED            ; GO TO DECIMAL FOR SCORE
70CE:18         234           CLC            ;
70CF:AD F0 70   235           LDA  DECCNT    ; INCREMENT DECIMAL
70D2:69 01      236           ADC  #$01      ;
70D4:8D F0 70   237           STA  DECCNT    ;
70D7:D8         238           CLD            ; GET OUT OF DECIMAL!
70D8:4C 32 70   239           JMP  CMND8     ; GO GET NEXT COMMAND


70DB:A0 BD      241 EMPTY8    LDY  #>MS6-CV1 ; GET EMPTY MESSAGE
70DD:20 E3 70   242           JSR  TEXT8     ; PUT ON SCREEN
70E0:4C 32 70   243           JMP  CMND8     ; GO GET NEXT COMMAND


70E3:           245 ;         *** TEXT GENERATOR ***


70E3:           247 ;   THIS USES THE SHORT FILE METHOD TO PUT
70E3:           248 ;   MESSAGES ONLY ON THE SCREEN.


70E3:B9 3B 71   250 TEXT8     LDA  CV1,Y     ; GET NEXT CHARACTER
70E6:F0 06      251           BEQ  DONE8     ; TEST FOR $00
70E8:20 F0 FD   252           JSR  COUT1     ; OUTPUT TO SCREEN
70EB:C8         253           INY            ; GO TO NEXT CHARACTER
70EC:D0 F5      254           BNE  TEXT8     ; AND REPEAT

70EE:60         256 DONE8     RTS            ; RETURN WHEN FINISHED


70EF:           258 ;         *** CARD COUNTER STASH ***


70EF:00         260 HEXCNT    DFB  $00       ; HEX COUNT FOR MACHINE
70F0:01         261 DECCNT    DFB  $01       ; DECIMAL COUNT FOR PEOPLE
```

PROGRAM RM-8, CONT'D . . .

```
70F1:           264 ;          *** SHUFFLER SUBROUTINE ***

70F1:           266 ;   THIS MODULE REARRANGES THE ARRAY CALLED CARDECK
70F1:           267 ;   AND WHOSE LENGTH IS STORED IN ARNUM.
70F1:           268 ;
70F1:           269 ;   THE RANDOM SUBROUTINE MUST BE PRESENT IN THE
70F1:           270 ;   MACHINE AND MUST BE PREVIOUSLY SEEDED AND
70F1:           271 ;   INITIALIZED.


70F1:AE 0E 71   273 SHUFFLR LDX   ARNUM      ; GET NUMBER OF SWAPS
70F4:CA         274         DEX              ; FOR ARRAY 0-51, NOT 1-52
70F5:20 7B 6F   275 NEXT8   JSR   REUSEN     ; GET RANDOM POSITION
70F8:A8         276         TAY              ; AND HOLD IN Y REGISTER
70F9:BD 13 72   277         LDA   CARDECK,X  ; GET FIRST FIXED VALUE
70FC:48         278         PHA              ; STASH TO JUGGLE
70FD:B9 13 72   279         LDA   CARDECK,Y  ; GET RANDOM NEXT VALUE
7100:9D 13 72   280         STA   CARDECK,X  ; REPLACE NEXT WITH FIRST
7103:68         281         PLA              ; JUGGLE BACK
7104:99 13 72   282         STA   CARDECK,Y  ; REPLACE FIRST WITH NEXT
7107:20 0F 71   283         JSR   EFFECT1    ; MAKE NOISE (OPTIONAL)
710A:CA         284         DEX              ; ONE LESS POSITION
710B:10 E8      285         BPL   NEXT8      ; REPEAT FOR EACH POSITION
710D:60         286         RTS              ; QUIT WHEN FINISHED


710E:           288 ;          *** SHUFFLER STASH ***

710E:34         290 ARNUM   DFB   52         ; NUMBER OF ELEMENTS IN ARAY


710F:           292 ;          *** SHUFFLER SOUND EFFECTS ***

710F:8A         294 EFFECT1 TXA              ; DECK SHUFFLING SOUND
7110:D0 02      295         BNE   NOZERO8    ; DISALLOW ZERO VALUE
7112:A9 01      296         LDA   #$01       ;
7114:0A         297 NOZERO8 ASL   A          ; SLOW IT DOWN!
7115:20 A8 FC   298         JSR   WAIT       ; DELAY, THEN FALL THROUGH
7118:A9 05      299 EFFECT2 LDA   #$05       ; NUMBER OF CLICKS PER WHAP
711A:48         300 NEXTWP  PHA              ; SAVE ON STACK
711B:2C 30 C0   301         BIT   SPKR       ; MOVE SPEAKER CONE
711E:A9 07      302         LDA   #$07       ; SET PITCH OF WHAP
7120:20 A8 FC   303         JSR   WAIT       ;
7123:68         304         PLA              ; GET CLICK COUNTER
7124:38         305         SEC              ;
7125:E9 01      306         SBC   #$01       ; AND COUNT DOWN
7127:D0 F1      307         BNE   NEXTWP     ;
7129:60         308         RTS              ; AND RETURN
```

PROGRAM RM-8, CONT'D . . .

```
712A:              311 ;          *** MESSAGE FILE ***


712A:              313 ;  WE'LL USE THE SHORT FILE METHOD HERE SINCE
712A:              314 ;  RANDOM ACCESS OF A FEW SHORT AND FIXED
712A:              315 ;  MESSAGES ARE NEEDED.
712A:              316 ;


712A:              318 ;          *** MESSAGE POINTERS ***


712A:00            320 CARVAL  DFB  >CV1-CV1  ; THESE POINT TO CARD VALUES
712B:04            321         DFB  >CV2-CV1  ;
712C:08            322         DFB  >CV3-CV1  ;
712D:0E            323         DFB  >CV4-CV1  ;
712E:13            324         DFB  >CV5-CV1  ;
712F:18            325         DFB  >CV6-CV1  ;
7130:1C            326         DFB  >CV7-CV1  ;
7131:22            327         DFB  >CV8-CV1  ;
7132:28            328         DFB  >CV9-CV1  ;
7133:2D            329         DFB  >CV10-CV1 ;
7134:31            330         DFB  >CV11-CV1 ;
7135:36            331         DFB  >CV12-CV1 ;
7136:3C            332         DFB  >CV13-CV1 ;


7137:41            334 CARSUIT DFB  >CS0-CV1  ; THESE POINT TO THE CARD SUITS
7138:48            335         DFB  >CS1-CV1  ;
7139:51            336         DFB  >CS2-CV1  ;
713A:57            337         DFB  >CS3-CV1  ;
```

PROGRAM RM-8, CONT'D . . .

```
713B:          340 ;     *** THE CARD VALUES ***

713B:C1 C3 C5  342 CV1    ASC          "ACE"
713E:00        343        DFB          X

713F:D4 D7 CF  345 CV2    ASC          "TWO"
7142:00        346        DFB          X

7143:D4 C8 D2  348 CV3    ASC          "THREE"
7146:C5 C5
7148:00        349        DFB          X

7149:C6 CF D5  351 CV4    ASC          "FOUR"
714C:D2
714D:00        352        DFB          X

714E:C6 C9 D6  354 CV5    ASC          "FIVE"
7151:C5
7152:00        355        DFB          X

7153:D3 C9 D8  357 CV6    ASC          "SIX"
7156:00        358        DFB          X

7157:D3 C5 D6  360 CV7    ASC          "SEVEN"
715A:C5 CE
715C:00        361        DFB          X

715D:C5 C9 C7  363 CV8    ASC          "EIGHT"
7160:C8 D4
7162:00        364        DFB          X

7163:CE C9 CE  366 CV9    ASC          "NINE"
7166:C5
7167:00        367        DFB          X

7168:D4 C5 CE  369 CV10   ASC          "TEN"
716B:00        370        DFB          X

716C:CA C1 C3  372 CV11   ASC          "JACK"
716F:CB
7170:00        373        DFB          X

7171:D1 D5 C5  375 CV12   ASC          "QUEEN"
7174:C5 CE
7176:00        376        DFB          X

7177:CB C9 CE  378 CV13   ASC          "KING"
717A:C7
717B:00        379        DFB          X
```

```
PROGRAM RM-8, CONT'D . . .


717C:              382 ;      *** THE CARD SUITS ***


717C:C8 C5 C1   384 CS0    ASC           "HEARTS"
717F:D2 D4 D3
7182:00         385        DFB           X

7183:C4 C9 C1   387 CS1    ASC           "DIAMONDS"
7186:CD CF CE
7189:C4 D3
718B:00         388        DFB           X

718C:C3 CC D5   390 CS2    ASC           "CLUBS"
718F:C2 D3
7191:00         391        DFB           X

7192:D3 D0 C1   393 CS3    ASC           "SPADES"
7195:C4 C5 D3
7198:00         394        DFB           X




7199:              396 ;      *** TEXT SCREEN MESSAGES ***


7199:C3 C1 D2   398 MS0    ASC           "CARD SHUFFLING DEMO"
719C:C4 A0 D3
719F:C8 D5 C6
71A2:C6 CC C9
71A5:CE C7 A0
71A8:C4 C5 CD
71AB:CF
71AC:8D 8D 00   399        DFB           C,C,X

71AF:A8 D3 A9   401 MS1    ASC           "(S)HUFFLE, (D)EAL, (R)EPLAY, (Q)UIT ?
71B2:C8 D5 C6
71B5:C6 CC C5
71B8:AC A0 A8
71BB:C4 A9 C5
71BE:C1 CC AC
71C1:A0 A8 D2
71C4:A9 C5 D0
71C7:CC C1 D9
71CA:AC A0 A8
71CD:D1 A9 D5
71D0:C9 D4 A0
71D3:BF
71D4:8D 8D     402        DFB           C,C
71D6:AD AD AD  403        ASC           "---> "
71D9:BE A0
71DB:60 88 8D  404        DFB           P,B,C,C,X
71DE:8D 00
```

PROGRAM RM-8, CONT'D . . .

```
71E0:C3 C1 D2   407 MS2    ASC        "CARD "
71E3:C4 A0
71E5:00         408        DFB        X

71E6:A0 C9 D3   410 MS3    ASC        " IS THE "
71E9:A0 D4 C8
71EC:C5 A0
71EE:00         411        DFB        X

71EF:A0 CF C6   413 MS4    ASC        " OF "
71F2:A0
71F3:00         414        DFB        X

71F4:AE         416 MS5    ASC        "."
71F5:8D 8D 00   417        DFB        C,C,X

71F8:A0 A0 A0   419 MS6    ASC        "   SORRY, DECK IS EMPTY!"
71FB:D3 CF D2
71FE:D2 D9 AC
7201:A0 C4 C5
7204:C3 CB A0
7207:C9 D3 A0
720A:C5 CD D0
720D:D4 D9 A1
7210:8D 8D 00   420        DFB        C,C,X
```

PROGRAM RM-8, CONT'D . . .

```
7213:              423 ;       *** DECK OF CARDS ***


7213:              425 ;   THE LOW BYTE OF EACH ENTRY IS THE CARD
7213:              426 ;   VALUE WITH X1=ACE, X2=TWO, XA=TEN, ETC.
7213:              427 ;
7213:              428 ;   THE HIGH BYTE OF EACH ENTRY IS THE CARD
7213:              429 ;   SUIT WITH 0X=HEARTS, 1X=DIAMONDS, 2X=
7213:              430 ;   CLUBS, AND 3X=SPADES.


7213:01 02 03  432 CARDECK DFB           $01,$02,$03,$04,$05,$06,$07
7216:04 05 06
7219:07
721A:08 09 0A  433        DFB           $08,$09,$0A,$0B,$0C,$0D
721D:0B 0C 0D


7220:11 12 13  435        DFB           $11,$12,$13,$14,$15,$16,$17
7223:14 15 16
7226:17
7227:18 19 1A  436        DFB           $18,$19,$1A,$1B,$1C,$1D
722A:1B 1C 1D


722D:21 22 23  438        DFB           $21,$22,$23,$24,$25,$26,$27
7230:24 25 26
7233:27
7234:28 29 2A  439        DFB           $28,$29,$2A,$2B,$2C,$2D
7237:2B 2C 2D


723A:31 32 33  441        DFB           $31,$32,$33,$34,$35,$36,$37
723D:34 35 36
7240:37
7241:38 39 3A  442        DFB           $38,$39,$3A,$3B,$3C,$3D
7244:3B 3C 3D




*** SUCCESSFUL ASSEMBLY: NO ERRORS
```

APPENDIX A

DIFFERENCES BETWEEN "OLD" AND "NEW" EDASM

Apple Computer's EDASM editor/assembler has recently been over-hauled and upgraded. There are now *two* new versions, one for DOS 3.3e, and one for ProDOS. Both are available in their respective toolkits in Apple's *Workbench* series.

The bottom line is that EDASM is now a first class, first rate macroassembler with just about all the bells and whistles anyone could ask for, including dual file editing, "library" module insertion, in-place assembly, and co-resident assembly. While the editor portion of EDASM remains as putrid as ever, you can simply use Applewriter IIe and WPL instead, doing "new way" editing as in chapter five.

Included with either EDASM is a new debugging tool called the BUGBYTER. The BUGBYTER includes a fancy upgrade of the old miniassembler, along with greatly improved single step, trace, and debug routines in a package that lets you do much more and *do it* much more quickly. For instance, there are single keystrokes to pick any screen mode, and you can use the game paddle to control debugging speed. You can even debug parts of your code at full speed and parts at slow speed. This is most handy for time critical routines. BUGBYTER is runnable anywhere in memory.

Very little was lost in going from "old" EDASM to "new" EDASM. With "new" EDASM, reserved labels normally include "A," "a," "X," "x," "Y," and "y," instead of just "A." You also *must* separate the op code and operand of SKP, ROL, ROR, ASL, LRS, and LST. Thus, a "SKP5" or a "ROLA" command will generate error messages under "new" EDASM. To use "old" EDASM source code with "new"

EDASM, spaces must be added between op code and operand of all these commands. Tab settings are also different in the "new" version.

Here's a list of the important changes and improvements to the *editor* portion of DOS 3.3e version of "new" EDASM. Note that anything you don't like about these features is easily gotten around by doing "new way" editing under Applewriter IIe instead.

Here goes:

1. The author of "old" EDASM was Randy Wigginton; the new author is John Arkley, who upgraded and improved Randy's original work.
2. The BUGBYTER is included, a tremendous improvement over the old miniassembler, single step, and trace routines.
3. System ID routines are now supplied and standardized, letting you configure your code for a II, II+, or IIe.
4. The work buffer is now 26,000 characters long, which is somewhat shorter than "old" EDASM. However, with "new way" editing under Applewriter IIe, your edit file can be 48,000 characters long.
5. The ASMIDSTAMP is restricted in its form so that real time clocks can be supported.
6. The FILE command now displays the slot and drive.
7. The manuals are greatly improved and now include tutorials.
8. The command level now automatically accepts either upper or lower case.
9. Combined upper and lower case is now standard on the Apple IIe. On older Apples, new commands of SETL and SETU are available for those Apples with a shift key mod and a lower case display. Commands of [E] (shift to lower case) and [W] (shift to upper case) are available for very old Apples without lower case. The screen will not be legible in lower case on these older machines.
10. Direct DOS commands using the "." prefix are not filtered for possible damage. In particular, ".SAVE" will plow the works.
11. You still cannot insert into the middle of your source code using "old way" editing. You have to use APPEND and then COPY. With "new way" editing, you can, of course, insert anything you want any place you want any time you want.
12. There is a new VOL command that goes along with SLOT and DRIVE that will return the current disk volume in use.
13. There is a new ADD command that lets you add text beyond a certain line number. Thus ADD 16 will add new lines *beyond* old line 16, compared to INS 16 which would insert new lines *before* old line 16.
14. The INSert or ADD modes can now be stopped with either a [D] or [Q].
15. A new REPLACE mode erases and then overwrites in one step. Before you had to DELete and then INSert.
16. There now is a recovery procedure to undo the NEW command. It is hairy to use, but it does exist.
17. A command of L43-6 lists six lines starting at line 43. Any time the second number is less than the first one, it is interpreted as "how many?".
18. The [R] command will relist whatever you last asked of [L].
19. A new command of SETD lets you change the delimiter from a ":". This lets you search and replace on a colon. Space or carriage returns are not allowed as delimiters.
20. You can now edit on both a range of numbers and a search string.
21. You can edit two files at once. The command of SWAP moves the two files between the "active" and "passive" editing buffers, sort of like a [Y] split screen in Applewriter IIe. The command of KILL2 deletes the "passive" buffer, similar to a "[Y]-N" in Applewriter IIe.
22. You can pick either 40 or 80 column operation with a "COL 40" or "COL 80" command.

23. There is now a simple way to undo the END command. Just set MAXFILES 5 and Call 3075.

Here are the major improvements in the *assembly* portion of "new" EDASM:

1. The DOS 3.3 version of "new" EDASM will not do an assembler listing to disk. You have to use the ProDOS version if you want to capture your normally printed assembler listing as a disk text file.

2. The trailer on an assembler listing now includes the date, line count, and remaining free space.

3. An "@" following an ASM command will suppress object code generation. This is handy for "quick looks" and finding potential errors.

4. The ASMIDSTAMP is no longer essential. On "old" EDASM, a FILE NOT FOUND error message was generated.

5. You can single step the assembly process by pressing the spacebar. Repeated spacebar hits do one line at a time. Pressing [ESC] on a 40-column screen lets you see the right half of the screen, or else switches back to the left half. Any other letter key resumes assembly at full speed.

6. Two *direct* keyboard commands override any imbedded LST ON or LST OFF commands. Use [N] to stop the listing, [O] to continue it.

7. The assembler will accept the tab key, [I], or the spacebar to enter a tab. This greatly eases the "tab problem" with "new way" editing.

8. The label in the label field is now called an IDENTIFIER.

9. The "a," "X," "x," "Y," and "y" labels are now reserved, in addition to "A." You can go to a lot of trouble to defeat this reservation if you have to. Good practice would also tell you to reserve "P," "p," "S," and "s" as well.

10. Macros are now available. These are disk based and are inserted when and as needed. Parameters can be passed back and forth between source code and macro.

11. A new operand of "*" is available that uses the present assembler program counter location. Intended use is to set aside specific positions in a page of memory. This can also be used to "pad" your way up to the next even page boundary.

12. You can now generate an absolute reference to a page zero location. To do this, put the EQU *after* the place in the source code where it first is needed. This is handy when you want to force an absolute long load, store, or whatever from an address on page zero, because of timing or code length considerations.

13. An upgraded OBJ command lets you assemble directly into the machine, without assembling to disk first. Tests are made to make sure there is no conflict with the assembly code itself. The combination of an "OBJ" command with an "ASM @" will directly assemble code into memory without generating any listing.

14. A new SW16 command will accept "Sweet 16" mnemonics. Three new commands have also been added to the original Sweet 16, which is a 16-bit pseudo interpreter. A compare, long branch, and subroutine long branch are now available. Use of Sweet 16 is usually shorter and simpler, but slower than doing your own custom 16-bit routines. One source of the new Sweet 16 code is EDASM itself. Just tear it apart using the "tearing method" of *Enhancing Your Apple II and IIe, Volume I,* (Sams 21822).

15. An undocumented X6502 command will apparently accept 65C02 mnemonics and, presumably, 65XC16 mnemonics as well. This command appeared in the preliminary documentation with a "we don't support this" disclaimer, but was dropped completely in the final manual.

16. New commands of ZDEF, ZREF, and ZXTRN are available that are

extensions of DEF. These forward-looking features require a linking loader that is not yet supported.

17. A new STR command works like ASC, only it includes a byte counter as its first character. Thus ASC gives you a text message, while STR gives you a text message preceded by the number of actual characters in the message.

18. A new DATE command reads the nine ASCII values stored at $03B8-$03C0 and enters them into the object code being generated. These locations usually hold the date portion of the ASMIDSTAMP.

19. A new IDNUM command reads the six ASCII values stored at $03C3 through $03C8 and enters them into the object code being generated. These locations usually hold the identity portion of the ASMIDSTAMP.

20. Conditional assembly has undergone a major overhaul. New commands of IFNE (not equal), IFEQ (equal), IFLT (less than), IFLE (less than or equal), IFGT (greater than), and IFGE (greater than or equal) are now available. A command of FAIL is also available for printing error messages.

21. A space must separate the op code and the operand on the SKP and LST commands.

22. Logical operators are now available, using the "↑" symbol for AND, "|" for OR, and "!" for EXOR. These operators work *only* on 16-bit arguments.

23. A new INCLUDE command stops the main assembly, assembles a source code module off disk, and then picks back up on the main assembly. This is most handy for inserting "mix and match" stock library routines.

24. Two commands of SBUFSIZ and IBUFSIZ let you adjust the size of your work areas for the original source code and the INCLUDE library module. See the manual for details. Changing buffer sizes is not normally needed.

25. A new MACLIB command tells the assembler that any "illegal" mnemonics are really the names of macro routines. Each macro routine is automatically done as if it was an INCLUDE command.

26. The "formfeed bug" has presumably been fixed, but it is still a good idea to force your own page breaks using the PAGE command.

27. A special column is available on the assembly listing to show branch destination addresses. Execution cycle times can also be optionally shown.

28. There are all sorts of new LST options. You can now separately turn off or on display of execution cycle times (C), generated object code (G), warnings (W), unassembled source code from bypassed conditional assembly (U), macro statements (E), alphabetic symbol tables (A), numeric symbol tables (V), or "six-across" symbol listings (S).

29. Standard tabbing values are different from "old" EDASM. Default tabs are now 16, 22, and 36, instead of 14, 19, 29. More than 80 columns may be needed for all the listing features and long comments. The simplest way to handle this is with 12 pitch on a daisywheel printer, or else use your own custom and "tighter" tab values. HINT: Keep your comments shorter than you did with "old" EDASM. This will help a lot.

30. New macro commands of "&0" and "&X" are available that control passing of parameters from the main source code to the macros. "&0" tells the number of parameters present in the operand field of the calling statement. "&X" keeps track of the number of times a macro is used. This allows the creation of local labels.

31. You can do co-resident assembly in a 64K Apple IIe, where the editor and assembler modules stay in the machine at the same time. An "*" following the ASM command will get the source file out of your machine, rather than off disk. This greatly speeds up the edit-assemble-

test round trip process. On short programs in certain areas of your machine, you can do both co-resident and in-place assembly at the same time. There are restrictions: You cannot use chaining, insertion, or macros when doing this, and your source code in the machine will get overwritten.

Finally, here are the differences between the ProDOS and DOS 3.3e versions of EDASM:

1. The ProDOS buffer is 37,000 characters long.
2. The ASMIDSTAMP is severely restrictive. It must be in DD-MM-YY format for clock compatibility.
3. A blank SBTL line still gets you the date.
4. The PFX command reads the current prefix. As is typical in ProDOS, a CAT command gets you a 40 column catalog, while the CATALOG command gives you all 80 columns. The CREATE command will generate a sub-directory.
5. The TYPE command lets you edit certain other file types, rather than just text files. You can also BLOAD, BSAVE, XLOAD, and XSAVE nontext files. The SYS command changes the type of source code file.
6. The EXIT command returns you to ProDOS BASIC. Commands of PTON and PTOFF turn the printer off and on, while EXEC will do a supervisory routine.
7. Time and date are automatically inset if a clock card is present. A TIME command is supported.
8. You can no longer do co-resident assembly. Preliminary ProDOS documentation did not support macros. Editing of two files at once also may not be supported.
9. You can route an assembler listing to diskette, instead of to printer, by using a "PR#6,ZORCHFILE" command.
10. There is a PAUSE command available to temporarily hold up assembly.
11. The error message on an aborted assembly is completely useless.

I personally despise ProDOS. Why? Because it is so unconscionably bloated, so user vicious, so buggy, and so incredibly poorly written. Nonetheless, if you must make an EDASM disk-based assembler listing (for "camera ready" print quality, typesetting, insertions, etc.), you will have to use ProDOS. The procedure is to take your DOS 3.3e text file, convert it with CONVERT, assemble to disk under ProDOS, and then CONVERT it back to the sane world.
Sigh.
Both ProDOS itself and the "new" versions of EDASM have numerous bugs in them. We will pass them on to you as we find out more about them.
Several specific bugs for now: The ProDOS routine of CONVERT can sometimes destroy a DOS 3.3e diskette. Seems a sector counter doesn't get incremented properly. Long filenames will often cause assembly problems. If it does not feel too much like assembling something, the ProDOS version of EDASM will simply kick sand in your face, instead of telling you what went wrong. That "ASSEMBLY ABORTED: LINE 0" message sure is friendly and helpful.
On either "new" version of EDASM, you will get error messages on a SKP5 or a LSTOFF, or an ASLA, and other places where "old" EDASM let you skip the space between op code and operand. Unfortunately, I did this just about everywhere in this book. Correcting the printed listings would most likely cause more grief than it would solve.

So, we have instead corrected all of the source code on the companion diskette.

Just remember to be sure and separate *all* op codes and operands with a space on "new" EDASM, and you should not have too much trouble.

Let us know about any other bugs as soon as you can.

APPENDIX B

SOME NAMES AND NUMBERS

ANTHRO DIGITAL SYSTEMS
Box 1385
Pittsfield, MA 01202
(413) 448-8278

APPLE ASSEMBLY LINE
Box 280300
Dallas, TX 75288
(214) 324-2050

APPLE AVOCATION ALLIANCE
721 Pike Street
Cheyenne, WY 82001
(307) 632-8581

A.P.P.L.E.
304 Main South
Renton, WA 98055
(206) 271-4515

APPLE COMPUTER
10260 Bandley Drive
Cupertino, CA 95014
(408) 996-1010

AVOCET SYSTEMS
804 South State Street
Dover, DE 19901
(302) 734-0151

BYTE
70 Main Street
Peterborough, NH 03458
(603) 924-9281

CENTRAL POINT SOFTWARE
Box 19730
Portland, OR 97219
(503) 244-5782

COMPUTER SHOPPER
Box F
Titusville, FL 32780
(305) 269-3211

CREATIVE COMPUTING
Box 789-M
Morristown, NJ 07960
(201) 540-0445

DENVER APPLE PI
Box 14767
Denver, CO 80217
(303) 429-4436

DECISION SYSTEMS
Box 13006
Denton, TX 76203
(817) 382-6353

DIABLO SYSTEMS
24500 Industrial Blvd.
Hayward, CA 94545
(800) 227-2776

GENERAL INSTRUMENTS
600 West John Street
Hicksville, NY 11802
(516) 733-3107

GTE ELECTRONICS
2000 West 14th Street
Tempe, AZ 85281
(602) 968-4431

HARDCORE COMPUTING
Box 44549
Tacoma, WA 98444
(206) 531-1684

HAYDEN SOFTWARE
50 Essex Street
Rochelle Park, NJ 07662
(800) 343-1218

HOWARD W. SAMS & CO., INC.
4300 West 62nd Street
Indianapolis, IN 46206
(800) 428-3696

INCIDER
80 Pine Street
Peterborough, NH 03458
(603) 924-9471

INFOWORLD
530 Lytton Avenue
Palo Alto, CA 94301
(415) 665-1330

INTERNATIONAL APPLE CORE
908 George Street
Santa Clara, CA 95050
(408) 727-7652

DON LANCASTER
Box 809
Thatcher, AZ 85552
(602) 428-4073

LAZER SYSTEMS
925 Loma Street
Corona, CA 91720
(714) 735-1041

LJK ENTERPRISES
Box 10827
St. Louis, MO 63129
(314) 846-6124

DAVID W. MEYER
600 Columbus Street
Salt Lake City, UT 84103
(801) 359-2790

MICROCOMPUTING
80 Pine Street
Peterborough, NH 03458
(603) 924-9471

MICRO INK
34 Chelmsford Street
Chelmsford, MA 01824
(617) 256-3649

MICRO LOGIC CORP.
Box 174
Hackensack, NJ 07602
(201) 342-6518

MICRO SCI
17742 Irvine Blvd.
Tustin, CA 92680
(714) 731-9461

MICROSOFT
10700 Northrup Way
Bellevue, WA 98004
(206) 828-8080

MICRO SPARC
10 Lewis Street
Lincoln, MA 01773
(617) 259-9039

MITEL
360G Leggett Drive
Kanata, Ontario K2K 1X5
(613) 592-5630

MOS TECHNOLOGY
950 Rittenhouse Road
Norristown, PA 19401
(215) 666-7950

MOTOROLA SEMICONDUCTOR
Box 20912
Phoenix, AZ 85018
(602) 244-6900

NEC ELECTRONICS
532G Broadhollow Road
Mellville, NY 11747
(213) 973-2071

NCR MICROELECTRONICS
1635 Aeroplaza Drive
Colorado Springs, CO 80916
(303) 596-5795

NIBBLE
Box 325
Lincoln, MA 01773
(617) 259-9710

PEELINGS
Box 188
Las Cruces, NM 88004
(505) 526-8364

QUALITY SOFTWARE
6660 Reseda Blvd.
Reseda, CA 91355
(213) 344-6599

RAK-WARE
41 Ralph Road
West Orange, NJ 07052
(201) 325-1885

ROCKWELL INTERNATIONAL
3310 Miraloma Avenue
Anaheim, CA 92803
(800) 854-8099

SAN FRANCISCO APPLE CORE
1515 Sloat Blvd.
San Francisco, CA 94132
(415) 556-2324

S-C SOFTWARE
Box 280300
Dallas, TX 75228
(214) 324-2050

SIERRA ON-LINE
36575 Mudge Road
Coarsegold, CA 93614
(209) 683-6858

SOFTALK
11160 McCormick Street
North Hollywood, CA 91603
(213) 980-5074

SOUTHWESTERN DATA SYSTEMS
10761 Woodside Avenue
Santee, CA 92071
(619) 562-3221

STELLATION TWO
Box 2342
Santa Barbara, CA 93120
(805) 966-1140

SYNERGETICS
Box 1300
Thatcher, AZ 85552
(602) 428-4073

SYNERTEK
Box 552
Santa Clara, CA 95052
(408) 988-5600

TEXAS INSTRUMENTS
Box 401560
Dallas, TX 75240
(214) 995-6611

THUNDER SOFTWARE
Box 31501
Houston, TX 77231
(713) 728-5501

WASHINGTON APPLE PI
Box 34511
Bethesda, MD 20817
(202) 332-9012

WESTERN DESIGN CENTER
2166 East Brown Road
Mesa, AZ 85203
(602) 962-4545

APPENDIX C

LABEL LISTS TO COPY

LABEL LIST FOR [   ]

DONE BY [   ]   ASSEMBLER [   ]

DATE [−   −]   SYSTEM [   ]

VERSION [   ]   [   ]

LABEL	EQU	LINE	DFB	VALUE	USE

NOTES _____   PAGE [ ] OF [ ]

_____   [   ]

_____

Index

COMPANION DISKETTE
AND VOICE HOTLINE

Don Lancaster and *Synergetics* have arranged to make available all of the source code and all of the object code shown in this book, along with lots of extra goodies in a crammed-full and fully copyable support diskette, in your choice of EDASM or S-C Assembler source code formats. The $19.95 price includes shipping and handling, as well as free voice hotline and support service.

Be sure to specify whether you want the EDASM or the S-C ASSEM-BLER version of this diskette.

You can order your companion diskette by using the card on the next page, or else directly from:

SYNERGETICS
746 First Street
Box 809
Thatcher, AZ 85552
(602) 428-4073

CONTENTS OF THE COMPANION DISKETTE

EMPTY SHELL.SOURCE
EMPTY SHELL
FLPRINT.SOURCE
FLPRINT
IMPRINT.SOURCE
IMPRINT
TIME DELAY.SOURCE
TIME DELAY
OBNOXIOUS SOUNDS.SOURCE
OBNOXIOUS SOUNDS
MUSICAL SONGS.SOURCE
MUSICAL SONGS
OPTION PICKER.SOURCE
OPTION PICKER
RANDOM.SOURCE
RANDOM
SHUFFLE.SOURCE
SHUFFLE
AUTO-DEMO
THE WHOLE BALL OF WAX
ENGINE
WHY RND AINT
WPL.NUMBER
WPL.UNUMBER
WPL.RENUMBER
WPL.TAB
WPL.UNTAB
MONITOR TIME DELAY
MULTIPLE DELAY FINDER
. . . plus a few more

RESPONSE CARD

☐   Please keep me informed of any updates and additions to the Assembly Cookbook.

My Apple is the _____ version.

The Assembler I use is _____

The RIPOFF MODULES that I want to see next are

_____

_____

_____

The PROBLEM that I now have is

_____

_____

_____

Please put any ADDITIONAL COMMENTS here:

_____

_____

_____

_____

_____

NAME _____

STREET_____

CITY _____ STATE __ ZIP _____

   voice phone _____

   data phone _____

DISKETTE

Please send me ____ copies of the 28 program, DOS 3.3 COMPANION DISKETTE to Don Lancaster's Assembly Cookbook, at $19.95 each, postpaid.

I understand this disk is fully copyable for my personal use only.

Send the   ☐ EDASM
            ☐ SC ASSEMBLER version.

Please also send me ____ autographed and postpaid copies of Don Lancaster's THE INCREDIBLE SECRET MONEY MACHINE, a complete guide to creating your own computer, tech, or craft venture, at $7.95 each.

☐   I enclose check for $ _____
☐   Please charge my VISA account number

      —   —   —
_____

   Expiration Date _____

   Signature_____

NAME _____

ADDRESS_____

CITY _____ STATE __ ZIP _____

Please, no purchase orders. We also cannot ship to a foreign address.

SYNERGETICS

BOX 1300
THATCHER, AZ 85552

FROM

PLACE
POSTAGE
HERE

SYNERGETICS

BOX 1300
THATCHER, AZ 85552

FROM

PLACE
POSTAGE
HERE

SYNERGETICS

BOX 1300
THATCHER, AZ 85552

SAMS.™

Sams Books cover a wide range of technical topics. We are always looking for more information from you, our readers, as to which additional topics need coverage. Please fill out this questionnaire and return it to us with your suggestions. They will be appreciated.

Please check the areas of interest:

1. CURRENT TECHNOLOGIES
   - ☐ Electronics
   - ☐ Circuit Design
   - ☐ Computers
     - ☐ Business Applications
     - ☐ Fundamentals
     - ☐ Languages _____
       Specify
       _____
     - ☐ Machine Specific:_____
       _____
   - ☐ Microprocessors
   - ☐ Networking
   - ☐ Servicing/Repair
   - ☐ _____
       Other

2. NEW TECHNOLOGIES
   - ☐ Fiber Optics
   - ☐ Robotics
   - ☐ Security Electronics
   - ☐ Speech Synthesis
   - ☐ Telecommunications
     - ☐ Cellular
     - ☐ Satellite
   - ☐ Video
   - ☐ Other_____
     _____
     _____
     _____

3. Do you ☐ own ☐ operate a personal computer? Model _____

4. Have you bought other Sams Books? Please list: _____
   _____
   _____

5. OCCUPATION
   - ☐ Business Professional _____
       Specify
       _____
   - ☐ Educator
   - ☐ Engineer_____
       Specify
   - ☐ Hobbyist
   - ☐ Programmer
   - ☐ Retailer
   - ☐ Student
   - ☐ _____
       Other

6. EDUCATION
   - ☐ High School Graduate
   - ☐ Tech School Graduate
   - ☐ College Graduate
   - ☐ Post-graduate degree

COMMENTS _____
_____
_____

(OPTIONAL)

NAME _____

ADDRESS _____

CITY _____ STATE _____ ZIP _____
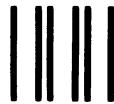
SAMS™ 22166   22331

Book Market Book

Book Mark

SAMS™ **SAMS**™

BUSINESS REPLY CARD	
FIRST CLASS PERMIT NO. 1076 INDIANAPOLIS, IND.	

POSTAGE WILL BE PAID BY ADDRESSEE

HOWARD W. SAMS & CO., INC.
4300 WEST 62ND STREET
P.O. Box 7092
Indianapolis, IN 46206

ATTENTION: Public Relations Department

SAMS

Assembly Cookbook for the Apple II/IIe

Your complete guide to using assembly language for writing your own top-notch personal or commercial programs for the Apple II and IIe.

- Tells you what an assembler is, discusses the popular assemblers available today, and gives you a list of the essential tools for assembly language programming.

- Covers source code details such as lines, fields, labels, op codes, operands, structure, and comments—just what these are and how they are used.

- Lets you find out the "new way" to do your source code entry and editing and to instantly upgrade your editor/assembler into a super-powerful one.

- Shows you how to actually assemble source code into working object code. Checks into error messages and debugging techniques.

- Includes nine ready to go, wide open ripoff modules that show you examples of some of the really essential stuff involved in Apple programming. These modules will run on most any brand or version of Apple or Apple clone, and they can be easily adapted to your own uses.

This cookbook is for those who want to build their machine language programming skills to a more challenging level and to learn to write profitable and truly great Apple II or IIe machine language programs.