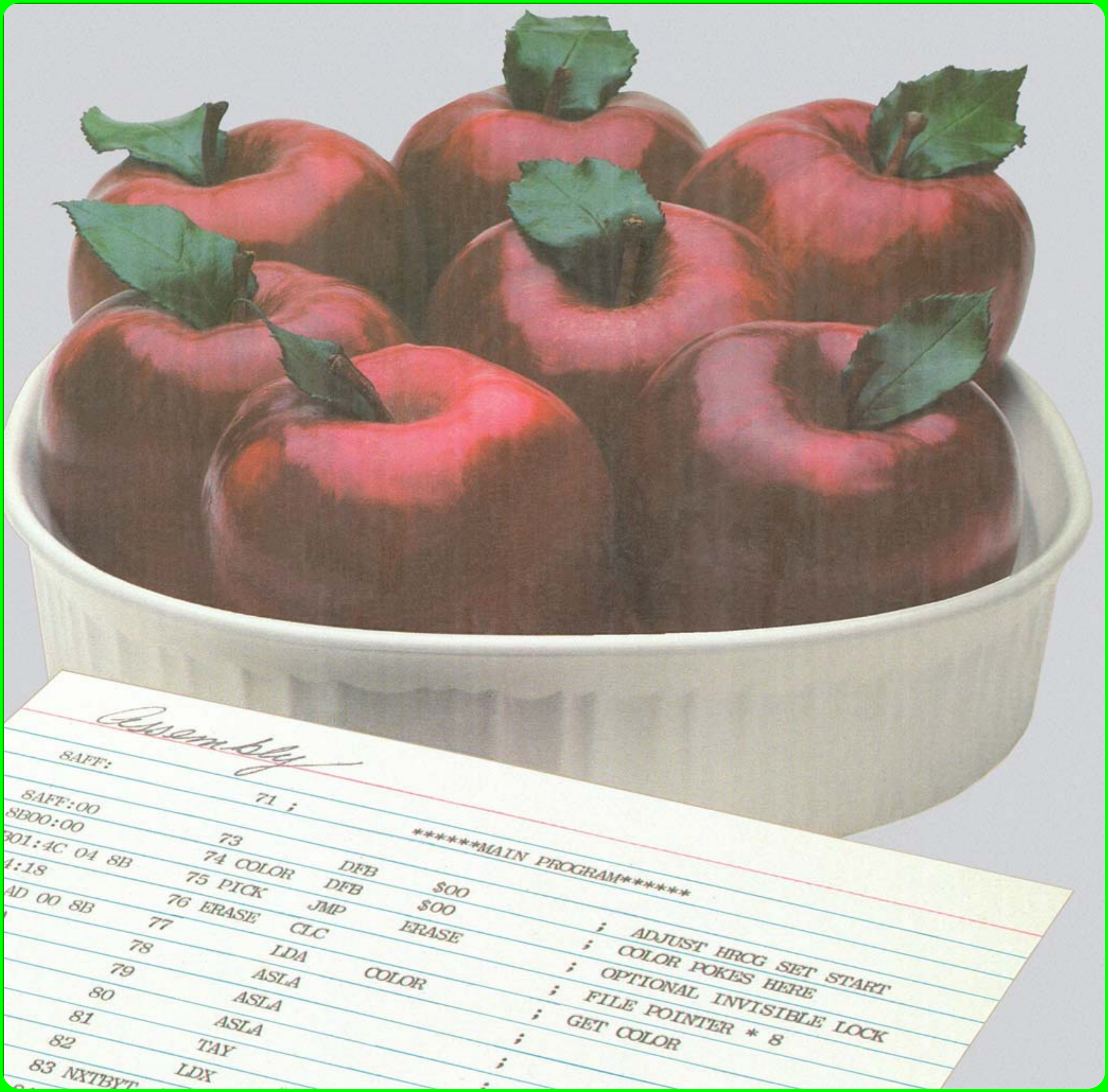
Assembly Cookbook for the Apple™ II/IIe

part one

Don Lancaster



```
SAFF:
SAFF:00                71 ;
8B00:00
8B01:4C 04 8B          73             ******MAIN PROGRAM******
:18                    74 COLOR  DFB  $00
AD 00 8B               75 PICK   DFB  $00
                       76 ERASE  JMP  ERASE     ; ADJUST HRCG SET START
                       77        CLC            ; COLOR POKES HERE
                       78        LDA  COLOR     ; OPTIONAL INVISIBLE LOCK
                       79        ASLA           ; FILE POINTER * 8
                       80        ASLA           ; GET COLOR
                       81        ASLA           ;
                       82        TAY            ;
                       83 NXTBYT LDX            ;
```

Assembly Cookbook for the Apple™ II/IIe

(part one)

by

Don Lancaster

SYNERGETICS SP PRESS
3860 West First Street, Thatcher, AZ 85552 USA
(928) 428-4073 http://www.tinaja.com
ISBN: 978-1-882193-16-5

0

Why you gotta learn Machine Language

Check into **Softalk** magazine's listing of the "top thirty" programs for your Apple. II or lie, and you'll find that thirty out of thirty of this month's winners usually involve mac:hine 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 lie 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 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 any 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.

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 direct machine programming.

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 many 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 a few 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 nonstandard 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 all in alphabetical order, you can play another compaction game by starting with a number that tells you how many of the beginning letters should stay the same in the next word, 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 lie 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 sing 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 known as **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 at least a few 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 lie 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 all the new cucumber cool 65C02 chips, which can allow a mind-boggling animation 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 SOX 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 reduce needed hardware. This is why 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 eeded 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 lie 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 some big 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 properly on first try.

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 when you go from the 6502 to a different microprocessor, and all the addressing modes and op codes will change.

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 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…

Assembly Cookbook for the Apple™ II/IIe

(part one)

Your complete guide to using assembly language for writing your own top notch personal or commercial programs for the Apple II and lie.

- **Tells you what an assembler is, discusses the popular assemblers available today, and details 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.**
- **Shows you 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, 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 up their assembly programming skills to a more challenging level and to learn to write profitable and truly great Apple II or lie machine language programs.

SYNERGETICS SP PRESS
3860 West First Street, Thatcher, AZ 85552 USA
(928) 428-4073 http://www.tinaja.com
ISBN: 978-1-882193-16-5