This enhancement works on all Apples and Apple knockoffs. There are additional monitor features available in the Apple IIe.

Enhancement

TEARING INTO MACHINE-LANGUAGE CODE

This method of breaking down and understanding someone else's Apple II machine-language program is — to say the least — unique. Here are complete details on how to rapidly "crack" both the form and function of any tough program. It takes only one-tenth of the time of orthodox methods.

TEARING INTO MACHINE-LANGUAGE CODE

Check into the top thirty Apple programs used today and guess what? At this writing, *thirty out of thirty* will run wholly, or at least partly, in machine language!

So, while BASIC language people are busy foisting computer literacy off onto the unwashed masses, and while Pascal people are stuffily trying to salvage what scant few shards remain of the once mighty computer science theocracy, and while FORTH people are out acting like spoiled brats . . . while all of this is happening . . .

Machine-language programmers are laughing to themselves all the way to the bank!

The evidence is in and it is overwhelming. Cash on the line. If you want to write a classic program or a best selling program, it **must** execute either wholly, or in part, in machine language.

Why?

Because machine language is far and away the fastest running, the most compact, the most flexible, the most versatile, and the one and only language that most fully utilizes all of the Apple's resources.

The only sure way to learn machine-language programming is to do lots of it on your own. But one thing that can help you a lot is to tear apart the winning machine-language programs of others to see what makes them tick.

You might also like to modify someone else's machine-language program to suit your own needs. Maybe you would like to find the scroll hooks in the HRCG *High-Resolution Character Generator*. Or perhaps you want to modify the original *Apple Writer* to output imbedded print format commands to your daisy-wheel. Or change *FID* to add your own "undelete file" command. Or maybe you have to modify a printer driver to handle HIRES graphics dumps. Or you might need some stunning animation. Or want to know what makes an adventure tick. Or whatever.

At any rate, if you brute-force attack someone else's machine-language program and if the program is more than a few hundred bytes long, chances are it will take you a very long time to crack it to the point where you think you understand it.

I'd like to share with you a method I use that will crack any unknown machine-language program astonishingly fast. The method does odd things odd ways, but ends up taking one tenth the time and one tenth the effort of any usual approach.

We'll assume you already know and have done some machine-language programming, and that the target program you want to tear into was written by an experienced and more or less rational programmer who didn't go very far out of his way to make things rough for you.

Let's see what is involved.

THE TOOLS

First, we'll have to put together a toolkit. You should have a tractor-feed printer along with some heavy white paper, preferably 20-pound paper. Naturally, you will also need a plastic *6502 Programming Card* and, of course, the *6502 Programming Manual*. The following listing gives a breakdown of the tools you will need to effectively tear apart machine-language programs.

You will also want all the usual *Apple* manuals, along with a copy of the *Apple Monitor Peeled*, and, if you can find one, a copy of the old red Apple Book. I'm also laboring under the delusion that you'll find *Don Lancaster's Micro Cookbook*, Volumes 1 and 2, of help (SAMS #21828 and #21829).

Try to get an Apple that has access to *both* an autostart ROM on a switchable plug-in card, and the old monitor ROM, without autostart, in socket F8 on the mainframe. This original ROM has the Trace feature, which was removed to make way for the autostart function. More importantly, the "old" ROM gives you the absolute control that is needed to stop any program at any time for any reason. Ads in *Computer Shopper* offer this ROM for $10.00.

Note that many newer programs will not let you drop into the monitor when you use the autostart ROM. Instead, they adjust the pointers so that they return to themselves on a system reset. Thus, an old ROM may be absolutely essential to let you view the target code. The Apple IIe may need custom EPROMs.

MACHINE-LANGUAGE TOOLKIT

( ) 48K Apple II, preferably with an old ROM in mainframe and switchable autostart ROM on plug-in card.

( ) Tractor-feed printer.

( ) Heavy white tractor paper.

( ) 6502 Programming Card.

( ) 6502 Programming Manual.

( ) All Apple manuals.

( ) Apple red book.

( ) Apple Monitor Peeled book.

( ) Lancaster's Micro Cookbook, Volumes 1 and 2.

( ) Roll of transparent tape.

( ) Case of page highlighters, in all available colors.

( ) Fine and regular felt-tip pens of matching colors.

( ) Serendipity scratch pad.

( ) What if? quadrille pad.

( ) A quiet workspace.

( ) The right attitude.

TOOLKIT SOURCES

( ) 6502 PROGRAMMING MANUAL
Rockwell International
Box 3669
Anaheim, CA 92803
Tel: 714-632-0950

( ) 6502 PLASTIC CARD
Micro Logic Corp.
Box 174
Hackensack, NJ 07602
Tel: 201-342-6518

( ) APPLE MONITOR PEELED
William Dougherty
14349 San Jose Street
Mission Hills, CA 91345
Tel: 213-896-6553

( ) THE MICRO COOKBOOKS
Howard W. Sams & Co., Inc.
4300 West 62nd Street
Indianapolis, IN 46268
Tel: 317-298-5566

( ) PAGE HIGHLIGHTERS
#2500A Major Accent
Sanford Corp.
Bellwood, IL 60104
Tel: 312-547-3272
(SEARS #3KX-3272)

If you really get into machine-language programming, this original firmware ROM is very, *very* useful. I suspect these ROMs may eventually become rare, but with 2716 EPROMs now under $5.00, you can easily clone your own by adding a simple $\overline{CS}$ adaptor to ROM socket F8.

You will want at least a 48K machine, and if there is extra RAM on plug-in cards, so much the better. The big advantage to having more RAM than the program needs is that you are free to add your own test and debug programs co-resident with whatever target program you are tearing apart. You should have both a cassette and at least one disk drive. The cassette can always save any image of any part of any program at any time, regardless of whether there is a DOS operating system there or not. Images on the tape can be split up and relocated as needed, letting you transfer them to disk at your convenience. The cassette can also let you introduce very small "test" and "hook" programs into the darndest spaces.

Now, off to the office supply. Get yourself a big roll of transparent mending tape—the kind you can write on. Then get two cases — yes, cases — of page highlighters. Throw away all the extra yellow ones, and get as many different colors as you can. Match each page highlighter with both a fine point and a regular felt-tip pen of the same color.

Don't underestimate the importance of these page highlighters. This method starts out real stupid like, but you will be astounded when the truth and beauty of what's happening leaps out at you halfway through. The highlighters are absolutely essential! Make sure these are the fat "see through" kind.

Get yourself some scratch pads as well. Label the little blank one "Serendipity" and the bit quadrille one "What if?".

You also have to have the right attitude, the right workspace, patience, persistence, curiosity, perversity, and a very distorted sense of humor for this method to work.

It is extremely important that you do everything that follows hands-on and *by yourself.* **Do not, under any circumstances, let someone else or the Apple help you with any tedious or dogwork parts.** The method relies heavily on your subconscious putting together the big picture and sewing up the loose ends. It can only do this if it has access to *everything* that the tearing-attack method needs. Do the dull stuff yourself!

THE FIRST RULE

What can we expect to find inside a machine-language program? The working code for sure. But, besides that working code, we need *files* that go with that code. In most longer machine-language programs, the files often take up far more room than the working code does.

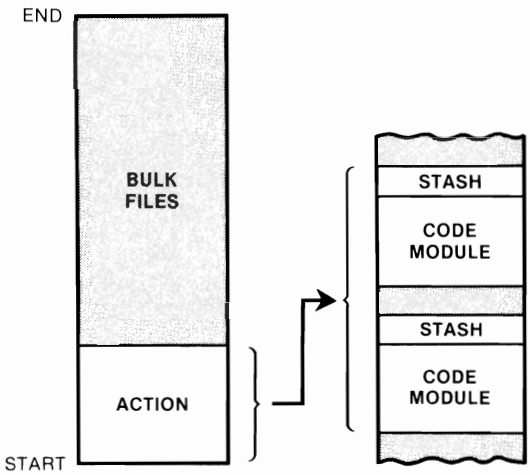


Fig. 3-1. A "typical" machine-language program.

Fig. 3-1 shows your "typical" machine-language program, which is just about as representative as your "typical" Apple owner or your "typical" rock. Anyway, we see that there are usually two main areas to a larger machine-language program. These are the *action* and the *bulk files.*

The action is the "real" part of the program that actually does things. The action, in turn, is made up of two different types of blocks. These blocks are called *code modules* and *stashes.*

A code module is a chunk of working machine-language code that does something. In most programs, most of the modules are subroutines, and are called as needed from a very short main program. The advantages of subroutines are that they break things down into small and understandable chunks and that they can be accessed from several places in the main program at once.

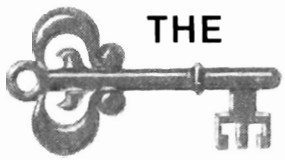
A stash is a short file that works directly with a module. The stash often follows immediately after the module that uses it. Typical stash entries might be a short ASCII string, a list of condition codes, or a table of indirect addresses. The stash holds values needed by the module that it works with.

The bulk files are usually much longer than the stashes. Bulk files normally sit off by themselves and usually follow the action. An example of a bulk file might be a high-resolution character set. The action controls how and when the character codes in the bulk-file character set go on the screen. In a medium-sized adventure, the bulk files may contain the map, the script, the objects, the responses, the rooms, and anything else unique to one particular story line. Only the bulk file has to be changed to change the adventure. The action can often stay the same.

In animated games or other programs that use the HIRES features, the bulk file may actually *be* the HIRES screen pages, or combinations of these pages with extra file space.

If you are into very fancy machine-language programs, the action may, in fact, be an interpreter acting as a special-use language. The bulk files will then contain commands that are run under the action's command interpretation. *Zork* is a classic example of this type of thing. In *Zork,* the action is a LISP-like interpreter specially written in compact and fast machine-language code.

The absolute key secret to tearing into machine-language code is . . .



Find out the STRUCTURE and the FLOW of any program, and most of the code will take care of itself!

So, never, *never,* **never** start taking apart machine-language code on a line-by-line basis. This is a total waste of time and will take forever.

Not to mention that it won't work anyhow.

The whole trick is to find out the *structure* of the program. Separate each module of the program and then separate each file from everything else. You'll find out there are very powerful hidden indicators that will leap out at you when you look for them. These indicators will very rapidly break everything down into simple, obvious, easy-to-understand, and self-documenting chunks.

Don't believe me? Let's try it and see. We'll use Apple's own HRCG *High-Resolution Character Generator* as a target program to show you how the method works and to illustrate key points. We'll go over the method in some detail. Later, we'll sum everything up in one checklist. HRCG is available on the DOS 3.3 TOOLKIT diskette, available from most dealers.

You'll get the most out of what follows by actually doing each and every step using your own copy of HRCG as we go along. Then try the method on a target program of your own choosing.

THE METHOD

Ready? Here we go.

GROK THE PROGRAM

You must be thoroughly familiar with what the program does and how it works before you start. Never try to crack a code until after you have used the program and really and truly know it.

For instance, there's absolutely no point in taking apart *Pyramid of Doom* to try and find the shovel. If you can't find the shovel, you just aren't cut out for Adventure. But, you just might want to tear into it to find the last treasure you need to replace the treasure you have to destroy to get past a certain —uh— inconvenience halfway up the pyramid. In no way will your first tearing into Adventure tell you the last treasure is in the dressing room, but you'll learn a lot about machine language and machine-language programs as you go along.

In the case of the HRCG, use the program and thoroughly explore all the alternate character fonts, and all the options of each and every mode of operation.

Know exactly what the program does before you try to tear into it.

One limit to this, though . . .

NEVER assume a program works in a certain manner or "has" to do something in an obvious way!

Thus, while you are learning how to use the program, and while you may think you have some good ideas on how the program works, *reserve judgement till later.* All your good ideas will invariably turn out to be 100% wrong.

If you can, watch others use the program and look into their reactions of how the program works and what it does. You may be missing something totally obvious. Rap with others as much as possible.

GO TO THE HORSE'S WHATEVER

Read every scrap of documentation that comes with the program, no matter how badly written or misdirected it may seem. Always ask around to see if the source code exists somewhere. Be sure to look into updates and revisions as well. It is infinitely easier to start with the original author's source code and work into the program, than to start with an unknown bunch of code and try to infer what the author had in mind in the first place.

If there is no documentation or if it isn't helpful, and if the original source code isn't available, *keep checking.* Perhaps others have torn into part of the code or have made modifications on their own that seem to work. Ask around at your club, school, computer store, bulletin board, or user group. If anything is available that seems to help, try it.

Anything else that can give you a clue to where the software author's head is and where he is coming from will be of great help. Maybe he publishes articles and stories. Maybe he has a series of programs out that can be of use.

A few moments of asking in the right places can save you months of time.

So, always check around.

HAVE A LIMITED GOAL

Any genuinely experienced programmer will admit to this rule . . . .

A long program is NEVER fully debugged nor fully understood. Nor can it ever be.

BELIEVE IT!

The entire DEW (Distant Early Warning) defense radar program was never tested. Not only was it never tested, the DEW program was so hopelessly complex that there was no possible way it could have been fully tested. Even if some test method existed, the probability of it passing any test was infinitely small.

A good and clean program simply has most of its remaining bugs fairly well hidden and fairly well out of the mainstream. This only happens after the ninth or tenth revision. But rest assured, there are definitely still bugs there, lying in "deep cover" and patiently waiting.

What this says is that the original programmer did not fully understand nor fully debug his program. If he says he has, he is either lying or else hopelessly naive. Now, if he didn't understand his own program, why should you?

Thus, a goal "to completely understand" some program is not only unreasonable; it is patently ridiculous. Instead, set yourself a reasonable and realistic goal for your first trip of tearing into machine code. Then, after you have set this realistic goal, simplify it till it is trivial. Then, simplify that. Then, think up some really dumb test of a small part of what is left. Something any idiot could hack. Maybe, just maybe, you will then be in the ball park.

For the HRCG, let's use the goal of answering "Where are the scroll hooks?" The HRCG obviously has some sort of scroll in it, since it moves characters up the screen. The scroll on the version I received is abrupt and chunky, so it can obviously be improved.

Or can it?

Maybe it's not so obvious. Why would such a good program have such an ugly scroll? These are name-brand people working on this and chances are they fumed and fretted over things quite a bit. Better stick with our original goal of finding the scroll hooks.

When you set your limited goal, don't become obsessed with it. The tearing method works by separating the *known* from the *unknown* as you go through the code. The method we will use demands a lot of apparently useless side trips.

Concentrate only on your goal and you may never get there.

FIND WHERE THE PROGRAM SITS

Before we can go on with our tearing attack method, we have to take time out for a rather long, but most essential side trip. Ready? Here we go . . .

Where is the machine-language program likely to sit? A glib answer is somewhere between $0000 and $FFFF, unless they are using memory mapping to go beyond 64K or are swapping things back and forth to the disk, or are using auxiliary memory on the Apple IIe. This assumes, of course, that the program is not self-modifying so that it changes itself through time.

Figs. 3-2 through 3-6 show us some places we can put a program. We can divide these into *low RAM, high RAM,* and *wherever.* Let's check these in more detail.

Low RAM

Low RAM is heavily used. As Fig. 3-2 shows us, low RAM goes from hex $0000 through $07FF, or memory pages Zero through Seven. Most of this space is reserved by the Apple for "system" uses. Let's check this out on a page-by-page basis . . . .

Page Zero is extremely valuable real estate for two reasons. The first is that the 6502 has a page Zero addressing mode that is shorter and faster than most

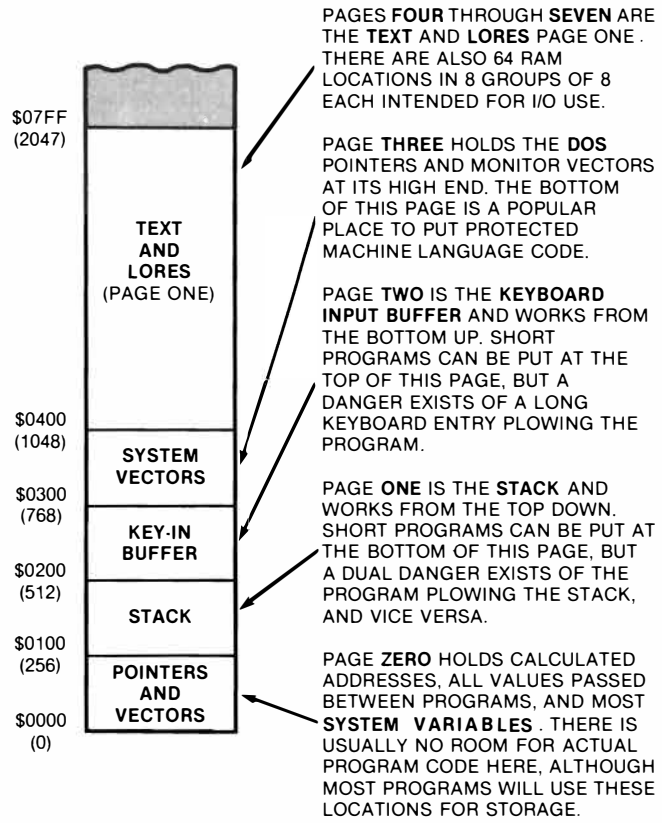


Fig. 3-2. Low RAM memory map.

other addressing modes. The second is that the two most powerful 6502 addressing modes — indirect indexed and indexed indirect — demand pairs of address locations on page Zero.

The Apple book shows how practically all of page Zero is used up one way or another by the monitor, the DOS, or either BASIC. For instance, the locations for the keyboard entry hooks and the print output hooks are stored as addresses on page Zero, as are the screen formatting controls that set the height and width of the display. Other important page Zero locations convert line numbers into the base addresses needed to hit a certain line of video.

We will see a list of these important page Zero locations shortly. The point here is . . .

> **Practically all programs need a few locations on page Zero.**
> **Some of these are used to pass values into the monitor, to BASIC, or to another part of itself.**
> **Other page Zero locations are used to hold calculated addresses for the indirect addressing modes.**

Thus, page Zero real estate is far too costly for program code. Instead, the available locations are used to pass values back and forth between the system and the target program, and to hold calculated address values.

Sometimes a target program will reassign page Zero locations for its own use. For instance, if the target program is fully in machine language, it can borrow

many of the locations "reserved" for Applesoft or Integer BASIC, since these locations will never be used. Monitor locations that serve oddball purposes can also be "redefined" provided that the monitor feature is never used, even by accident.

Occasionally a very short machine-language sequence can be crammed into low values on page Zero, as was done with the original tone subroutine in the old red book. Even this got you in trouble when you switched to Applesoft. So, putting programs on page Zero is both dangerous and dumb, but it can be done.

Another dangerous place to put programs is on page One. Page One is intended to be used for the *stack.* The 6502 uses a single stack that starts at location $01FF and builds down. This stack is shared by the monitor, the operating system, and the program itself. Important uses of the stack are to store the return address of a subroutine call and both return address and processor status on an interrupt. Advanced programmers might also use the stack as a temporary stash of a value or two, or might even manipulate the stack to alter the program flow.

The stack rarely gets below $0180 in normal use. It is usually possible to put a very short machine-language program in locations $0100 through $017F. This is dangerous, since the program can plow the stack and vice versa, if either gets too long.

Page Two is normally used as a *keyboard buffer.* Key entries start at $0200 and build their way up. The average number of keystrokes stored is fairly low, and you can sometimes cram a small machine-language program on the top of this page. Once again, you are asking for trouble since too long a keyboard entry will plow your program.

One sneaky and ugly trick that a programmer can pull is to put some relocation or protection code starting at $0200. This code must be used before any keys are hit, and is thus very difficult to read. The code will, of course, get destroyed as soon as any keys are entered.

Most of page Three is available to the machine-language programmer. There are some DOS jumps and system vectors on the high end of this page. The vectors control the reset, interrupt, autostart return, breakpoints, Applesoft "&", and nonmaskable interrupt jumps.

Thus, you are free to use the first 150 or so locations on page Three for your machine-language program. This turns out to be a favorite stash for short programs, since this area is automatically protected from either BASIC.

Unfortunately, everybody and his brother crams just about everything they can think of in here, and you can often have two parts of a program, each of which needs a different machine-language code, both trying to use this space. For instance, a printer driver may be placed here by one program and a screen dump by another. Try to combine the programs, and you have a turf fight.

If you have a longer machine-language sequence, you can sometimes combine the top half of page Two continuously with the bottom half of page Three. Again, you have to be careful not to get bumped by a long keyboard entry and to be sure you don't, in turn, bump into a DOS hook or other pointer.

Memory pages Four through Seven are the page One text screen and page One LORES screen. The only difference between traditional text and LORES is that, in text, the stored code goes through a hardware character generator while, in LORES, the same code is directly bit-by-bit converted into a stacked pair of colored blocks.

It seems kinda dumb to try and put machine-language code onto the display pages. First, you will probably see it and it will look ugly. Secondly, any scrolling or screen clearing will destroy the code. Nonetheless, in a program that does all its work in HIRES, this space is theoretically available.

There are some sneaky RAM locations stashed here and there on pages Four through Seven that are *not* displayed and are *not* erased by a properly done scroll or clear. There are 64 of these locations. These are normally intended for use by the I/O slots and have intended assignments.

If you really want to be tricky, you can use these spaces any way you want to, provided there is no I/O access to the same location. This is one of the better hiding places for disk verification codes and other sneaky stuff.

Summing our low RAM up, you have a few locations on page Zero available to you that are usable to pass values to the monitor or to save calculated addresses. The low end of the page One stack and the high end of the page Two keyboard buffer can be used for short programs or subroutines, but use of these areas can be dangerous. Most of the bottom of page Three can be used for a machine-language program. This space is very popular but it can cause conflicts between programs. Finally, pages Four through Seven are the page One text and LORES display and are not normally available for program storage, except for some 64 hidden locations that are normally reserved for input and output.

High RAM

As Fig. 3-3 shows us, the high RAM runs from $0800 up through the top of installed RAM. In a 48K machine, high RAM goes from $0800 through $BFFF. This area holds the usual locations where longer machine-language programs are placed.

How much of high RAM is available for your use? It all depends on what other features you are going to run along with your program, and what minimum size Apple you want the program to run on.

We will assume that the target program needs a full 48K. Extra RAM is now so cheap that practically all Apples either arrive with full RAM or are soon filled. With those new 64K RAM cards, most Apples will soon have bunches of extra memory on top of what used to be "full." A machine with a mere 48K of RAM will soon be at poverty level.

At any rate, if you decide to use text page Two or LORES page Two, locations $0800 through $0BFF have to be set aside and protected. Use of this text page is relatively rare.

If you want to use HIRES page One for graphics, sprite animation, or multifont text displays, then locations $2000 through $3FFF have to be reserved. Use HIRES page Two and you will also have to reserve locations $4000 through $5FFF. These locations hold an image of what goes on the screen and, thus, are not available for both display and program use at the same time. You will sometimes use both pages at once for effective and fast animation or to double graphics resolution.

While there are a few unused RAM locations on these HIRES pages, these locations get plowed every screen reset or color change. Thus, they are not safely usable except as a very temporary stash.

We will note in passing that if the HIRES pages are not used, and you put code in this area, you can actually *watch* the code executing by switching to HIRES while the program is in action. This can be a very powerful snooping tool. Watching a program run its own code gives you a new window into what is happening. You can also watch code working on LORES page Two, but this is a much smaller area and not nearly as useful.

If you are using standard DOS, the space from $9600 through $BFFF is normally saved for the DOS system. You can sometimes "borrow" a DOS file or two and stuff a short machine-language sequence into a small portion of this protected space.

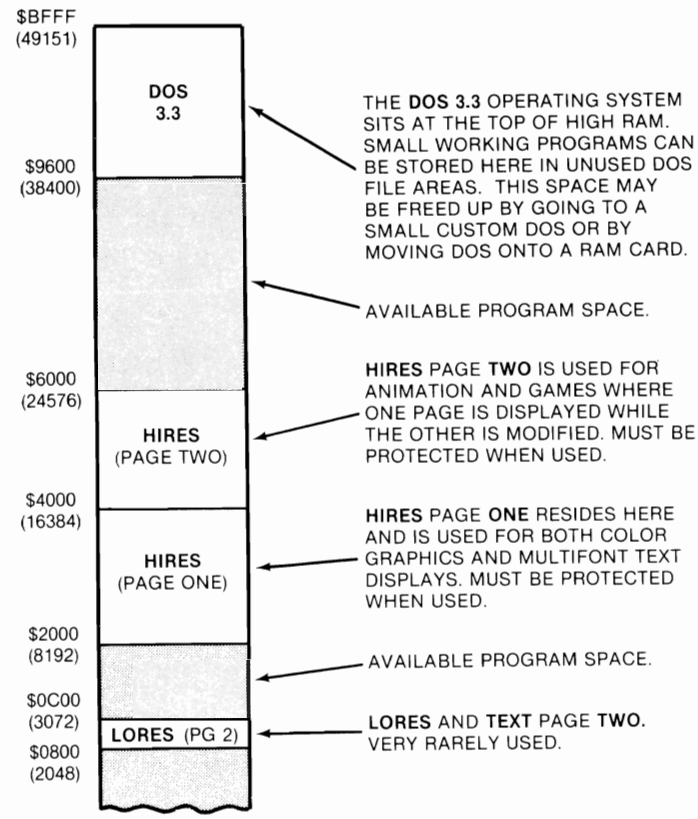


Fig. 3-3. High RAM memory map.

A lot of programs provide their own smaller and simplified versions of DOS. This gives a measure of copy protection and makes more room for the rest of the program.

Thus, a machine-language program could go from $0800 to $BFFF. Subtract the range $9600 through $BFFF for DOS at the top, the range $4000–$5FFF for HIRES page Two, the range $2000–$3FFF for HIRES page One, and, if used, the range $0800 through $0BFF at the bottom for text and LORES page Two.

Many machine-language programs start at $0800 and work their way upwards as needed. If they are about to crash into the HIRES pages, they skip above HIRES and continue as far as they have to.

Combining programs

Things get much more complicated if machine-language subroutines have to interact with Integer or Applesoft BASIC programs. Each BASIC language works differently and needs a different way to "protect" an area for its machine-language routines. The protection is needed to keep the BASIC from overwriting the machine code and vice versa. Fig. 3-4 shows us more detail.

In Integer BASIC, HIMEM is a high-memory pointer that points to the end of the Integer program. The program starts at HIMEM and builds its way downward. Every new program line gets put in its place, automatically moving everything else down and leaving you with the end of the program listing at HIMEM. String variables start at the low-memory pointer LOMEM and build their way upwards.

The usual way to tie a machine-language program into Integer BASIC is to start the machine-language sequence at $0800 and set LOMEM to at least one space above the end of the machine-language code. This LOMEM can be set as the

first instruction of an Integer BASIC program. It takes an "illegal" command, but it is easily done with a single POKE command. Should you also be using the HIRES pages, you still would start your machine-language program at $0800, but

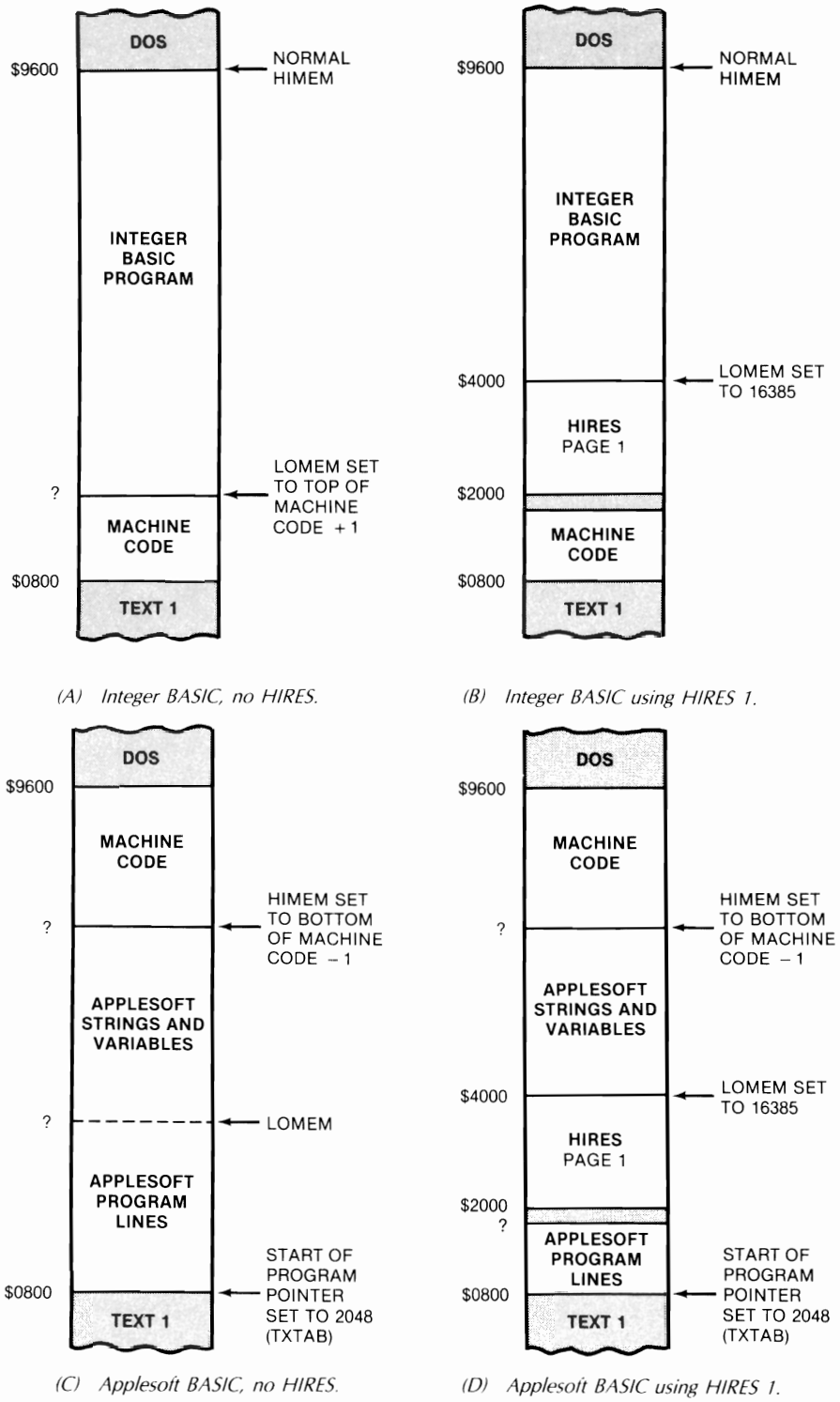


Fig. 3-4. Usual ways of combining BASIC and machine-language programs. Note that machine code goes *above* Applesoft or *below* Integer.

you would most likely reset your LOMEM pointer to one location above the highest HIRES screen location needed. This is shown in Fig. 3-4B.

Applesoft does things quite differently than Integer Basic. Applesoft programs start at a start-of-program pointer TXTAB and build their way up, while the string variables start at HIMEM and work down.

It is not normally possible to change the start-of-program pointer *during* a program since the program is already located in memory and is not movable. Thus, while you can, in theory, put a machine-language program below this pointer, the only way to do it is to change the start-of-program pointer *before* you load your final Applesoft program.

Note that this start-of-program pointer is not LOMEM! It is called TXTAB and sits at $0067 (low) and $0068 (high). LOMEM in Applesoft is actually in the middle. LOMEM points to the beginning of the variable space and often marks the end of the program lines.

You will usually put your machine-language program above Applesoft by setting HIMEM before you run your Applesoft program. HIMEM may also be set early in the program. Details on this are shown in Fig. 3-4C.

For more program room, you also have the option of setting HIMEM to one less than the start of your machine-language program, and LOMEM to one more than the highest HIRES location in use. The start-of-program pointer remains at $0800. This lets you put program lines from $0800 up through the start of the HIRES page, and place the strings and variables from the top of the HIRES space to the bottom of your machine-language code. This is shown in Fig. 3-4D.

So, we see that machine-language programs running with Applesoft normally go *above* HIMEM, while machine-language programs running with Integer BASIC normally go *below* LOMEM.

You can also play all sorts of pointer games to tow a short machine-language sequence along *inside* an Integer BASIC or Applesoft program. One way you can do this is to put the machine-language stuff *between* two BASIC statements. The parsed code on the first BASIC statement is then altered so it jumps *over* the machine-language part to get to the next expected instruction. These pointer schemes are tricky and really get hairy if you make any changes, but some authors use them to "protect" their programs or "hide" their fast machine code. The advantage of this is that you can use one cassette loading to enter both machine and BASIC codings. With a disk it is much simpler and saner to let one program load the other one by using a second disk command.

Mainframe RAM usually only goes up to 48K. What is in the other 16K of our 64K Apple? Figs. 3-5 and 3-6 complete the picture for us.

There are sixteen pages located from $C000 through $CFFF that are reserved for I/O. As Fig. 3-5 shows, the bottom half page ($C000 to $C07F) is used for all the screen switches, the push buttons, the paddles, speaker, cassette, keyboard entry, and the keyboard strobe. The next half page ($C080-C0FF) is used to pass address locations to each slot. There are sixteen locations reserved for each slot one through seven.

Above that, we see seven location blocks that are one page of 256 words each. These usually will hold the "control" PROM or ROM for a given card and are addressed as shown. A final 2K space is reserved from $C800 through $CFFF that can be used by *any* I/O slot that wants it, as long as all the slots take turns, and only one slot is active at a time.

There is usually very little RAM in the I/O space. These locations are important, though, for they are how we control the on-board things like the screen modes, speaker, paddles, keyboard, and so on. They are also the way we interact with any working card. If a plug-in card is involved with the code you want to tear into, you will have to pin down exactly what codes goes where.

If we now turn to the uppermost 12K of address space on the Apple, we see that there are six ROM sockets on the Apple mainframe. Each socket can hold a 2K×8 bytewide ROM or RAM. Fig. 3-6 shows us the usual setup for Integer BASIC or Applesoft machines. A 2K monitor ROM needs the top or $F8 socket. There are two possible monitors, the old or *absolute reset* one, and the newer *autostart* one.

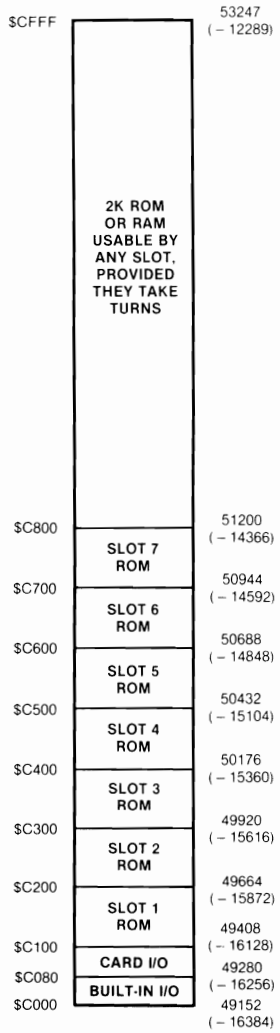
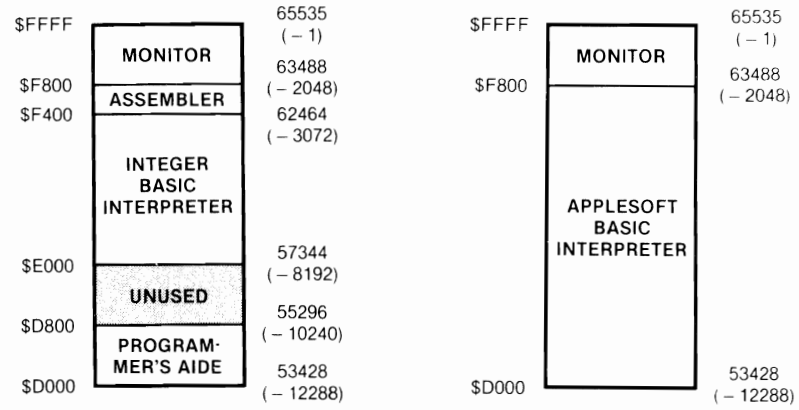


Fig. 3-5. I/O map.

Continuing down our ROM sockets, Applesoft uses the bottom five, while Integer BASIC uses the middle three, along with an optional *programmer's aide* that fits in the bottommost or "D0" socket. The uppermost Integer ROM at "F0" also holds the extremely useful mini-assembler code, along with the old floating-point package, and the "Sweet 16" 16-bit machine pseudocode. None of these machine-language test and debug features are available in the Applesoft ROMs.

This area is all ROM and cannot normally be written to. But the locations in this area are useful to interact with the monitor or either BASIC language.

The entire top of the machine can be bypassed by any plug-in card through the $\overline{\text{INH}}$ line. This can let a plug-in ROM card give you the switched choice of either BASIC, or it can let a RAM card do darn near anything it wants to, including running other languages, holding DOS, or giving you extra RAM space.



(A) Integer BASIC. *(B) Applesoft.*

Fig. 3-6. High ROM maps. A plug-in ROM or RAM card can deactivate these and substitute its own code.

Note that many software programs placed on RAM cards may deny you *ever* gaining access to the monitor ROM in mainframe socket $F8. This can make intercepting a running program rather tricky.

Many machine-language programs will start at $0800 and work their way upwards, but you can expect any program to go just about anywhere, depending on what other resources of the Apple are being tapped.

One ultrasneaky trick is to start your machine code at the bottom of the keyboard buffer at $0200, with a jump, and then run up through everything in between there and the end of your machine-language program. This neatly hides the "real" starting address of your program and also gives you an attractive page One text or LORES display while the rest of the program is loading.

You must, of course, find out where the program is before you can attack it. Let's start with a very obvious fact . . . .

OBVIOUSLY

You cannot tear a program apart that is not already in the machine and capable of running.

What this says is that any program that uses a disk may not have that part of the program in which you are interested sitting in the machine at any given time. This rule also says that any program must be placed in the machine exactly where it normally will run, and it must be started off on exactly the first instruction location.

So, be sure you have that part of the program that you want to analyze in the machine when you attack it.

The other side of the coin has the good news . . .

BUT, THEN AGAIN

At any given time, any working program MUST have everything it needs in the machine so it can continue.

So, if there is no disk whirring between where you are and what you want to analyze, it all has to be there in the Apple somewhere, somehow.

But, where is where?

You must pin down all of the exact locations a target program uses before you can tear into it. There are at least four good ways to do this . . . .

FINDING PROGRAM LOCATIONS

( ) Read the instructions.
( ) Ask DOS to tell you.
( ) Infer from use.
( ) Empty, then fill the machine.

The first and most obvious way is to see if the author didn't tell you somewhere just exactly where the program sits. For instance, the loading instructions for the *Adam's Adventures* 0–12 tell you these go from $0800 through $57FF and that the starting point is $0800. Being told ahead of time where the program starts and resides is the easiest and best method, so always look around carefully for loading information.

The second way to find where a machine-language program goes is to let DOS tell you. On a 48K machine BLOADed under standard DOS 3.3, the starting address ends up in $AA72 (low) and $AA73 (high). The program length is stashed in $AA60 (low) and $AA61 (high). After loading, you reset, do a call −151 to get into the monitor, and, then, inspect these locations. The old monitor ROM might be needed to force reset back into the monitor.

DOS can also give you some hints. If you can read the catalog, the type of file and its length should be obvious. Even listening to the number of track clicks during a load should tell you something about how long the program is and which disk tracks it lies on. Take off the disk drive cover, and you can actually *watch* the drive move from track to track. With some practice, you will be surprised how much this can tell you. This process, when formalized, is called *boot tracing.*

where the program sits from what it has to do and what it has to interact with. Our HRCG gives us a good example here. We can't directly find where HRCG sits since it is an "R", or relocatable, rather than a "B", or binary file.

But, the Applesoft Toolkit book tells us HRCG fits under DOS and moves HIMEM down to protect itself and its alternate character fonts from Applesoft incursion. There's a simple and easy-to-use BASIC program called LOADHRCG that comes with the HRCG program. In it is a variable called ADRS which equals HIMEM. Run this one with no alternate character sets, and we see that ADRS ends up as $8DFE. Run it with one alternate character set, and HIMEM moves three pages lower to $8AFE. Two alternate sets and HIMEM drops three more pages lower to $87FE, and so on. This special example is shown in Fig. 3-7.

So, by inference, HRCG sits from $8DFF through $95FF. This will include the HRCG action and the bulk file used for character set Zero, the default ASCII set. Other character sets build downward three pages at a time, with the lowest-numbered set on the bottom and the highest set always at the top, again as shown in Fig. 3-7.

You can find this out on your own by carefully studying a printout of the LOADHRCG Applesoft program and then doing loadings and finding the value of ADDR, otherwise known as HIMEM. The same study should show you how the alternate character sets are filled in.

The final method of pinning down a large program works even if all other methods fail, and should be used as a check even if you are absolutely sure where the target program sits. This final method is a sledgehammer. You empty the machine completely, and then refill it only with your target program. Then, you casually flip through memory, a page at a time, till you find the program. The next tearing step gives us full details on this.

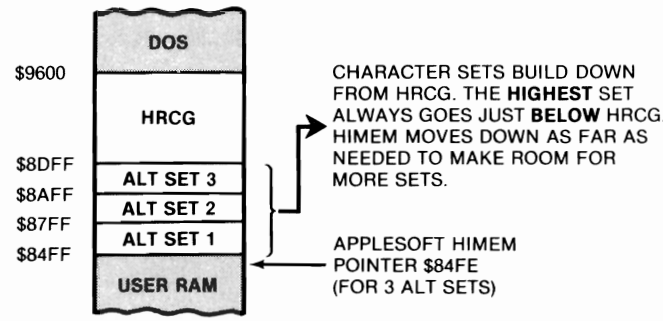


Fig. 3-7. Location of HRCG program and alternate character sets in a 48K Apple II.

We have now seen how an Apple's memory is arranged and the methods we need to use to find where a program sits. Let's now return to the mainstream of our tearing attack.

You can use any method you like to pinpoint exactly where the target program lies. Try reading instructions, and then try letting DOS tell you. Then, try inference from what the program does and how it interacts with the Apple. If none of that works . . . .

EMPTY THE MACHINE

There is nothing more infuriating than to find out you are really analyzing interpreted BASIC code left over from last Tuesday's 4 AM breakout game, instead of your target program.

To prevent this from happening, you will want to completely and absolutely *empty* your machine of everything old and unneeded before you begin. There are two very good reasons for this. One is that you won't be wasting your time analyzing something that is not part of your target program. The second is that an empty machine that has just been filled is one sure way to find or verify the location of your target program.

You should always clear your Apple of old stuff before attacking a target program. But, how do you empty a machine?

Even a just repowered Apple will come up with random garbage in most all of the RAM locations. The trick is to load each and every memory location with an obvious value that is very easy to spot, particularly when it is scrolling by. The value $00 is dangerous since it is also a Break command, and it is hard to read on the fly. I use the value $11 instead. On a listing, you get an unmistakable string of continuous lines on anything that is still empty. This pattern is readable even during an abrupt scroll.

The following steps show us how to empty your Apple. It's very easy to do from the monitor. You put a $11 somewhere and, then, move it as far up in memory as you want, recopying it over and over again. If you are using DOS 3.3, you should empty locations $0220 through $03CE, and $0800 through

$95FE. Be sure to empty your machine *after* booting DOS. Do things the other way around and the DOS boot code will return to haunt you.

If you are not using DOS, then you can go ahead and empty $9600 through $BFFF as well. You might also like to empty page One from $0100 to $0180. But, don't try to empty page Zero, the top half of page One, the first few locations of page Two, the top of page Three, or anything above $C000. Erasing any of these locations will bomb the machine or cause other problems.

To empty your Apple, put an "empty" symbol in some location. Then, use the monitor to move a block of memory—starting at that location and moving up by one.

A good empty symbol is "$11".

A. **To empty user RAM except for DOS:**

```
*0800: 11 < cr >
*0801 < 0800.95FEM < cr >
```

B. **To empty all user RAM:**

```
*0800: 11 < cr >
*0801 <0800.BFFEM< cr >
```

C. **To empty most of pages $02 and $03:**

```
*0220: 11 < cr >
*0221 < 0220.03CEM < cr >
```

To get into the monitor from either BASIC language, do a CALL −151. Once again, do not try to empty page Zero, the top half of page One, the first few locations on page Two, page Three above $03CF, or anything above $C000.

Some target programs will try to prevent you from ever going into the monitor. Switch to the old (nonautostart) monitor ROM if this happens.

When your machine is empty, snoop around everywhere to see what it looks like. From the monitor, do a 0800.BFFF < cr > and watch the "elevens" go streaming by.

You'll next want to load and verify the locations of the HRCG program from $8DFF through $95FF. Try adding alternate character sets, one at a time, and see what happens.

Always start with an empty machine and always return to one anytime you get confused as to what is happening.

LIST THE PROGRAM

After you have emptied the machine and loaded your target program, go ahead and list it. Make two copies on the heaviest white tractor paper you can find. You list a program from the monitor by typing the starting address and, then, the character "L" eighty times and, then, a <cr>. Each L command gets you twenty lines of disassembled code. Use too few L's and you will have to retype them in the middle of your listing. Too many and you simply hit RESET when you get to the end of the target program.

Keep three clean white pages before and after the listing. Do NOT take the listing sheets apart. Instead, carefully reinforce every tear line, tractor holes and all, with transparent tape. Actually, you would be best off having a welder transcribe a copy of the listing by burning it into quarter-inch steel plate for you.

No matter how rugged you make it, it won't be enough. The object here is to keep the listing in one piece and legible after handling and rehandling over and over again. So, don't spare the tape.

Label the top sheet with the name of the target program and the date you started attacking it. Don't forget the year and version number. The second copy is a backup to be used when the first one falls apart or gets totally illegible.

You will also want to make two copies of a hex dump of the target program. For HRCG, you get in the monitor, type 8DFF.95FF, reach over and move the printer paper up a space or two, and, then, hit <cr>. Incidentally, on both the listings and the hex dump, use the printer's skip-over-margin feature if you have it available.

Most of our tearing apart will be done on the listing sheets. The hex dump sheets will sometimes show us a pattern in a file or will give us some other pictorial information or other visual clues that can be of enormous help.

Yes, you might have to list and hex dump the entire machine for really fancy programs, and this will take bunches of paper and, maybe a ribbon or two. But this isn't nearly as bad as it seems, and it must be done if you are to crack the program.

Well, we finally have completed our preliminaries. It sure took a long time to get here. Now the fun starts. Ready?

SEPARATE THE ACTION FROM BULK FILES

Carefully look at your listing. Not for detail, but for overall vibes. Anytime you think something may be helpful, jot it down on one or another of the pads.

But, once again, do not jump to conclusions and do not attempt to analyze any part of the code in detail. At this stage in the game, we are interested only in the flow and pattern of the big picture.

The first thing we want to do is isolate the action so that we can work with it separately. As you go along, you will gain a feel for what I call "rational" code. Rational code has a flow to it, with reasonable commands used in reasonable ways. At this point, we don't want to pass judgement nor force conclusions as to what is which. But see if you can't separate obviously "rational" code from everything else.

Now, we told our lister to list—assuming that it would be handling working machine-language code. The lister will also try to list a file, or random garbage, *as if it was rational code.* So, we can expect lots of visual clues as to whether we are working on real code or file values. Here are some sure signs . . .

FILE CLUES DURING A LIST

( ) Lots of question marks.
( ) Break commands ($00).
( ) Dumb repetition.
( ) Rare commands in odd mixes.

The question mark means that the lister thought it had found an illegal op code, something that the 6502 microprocessor does not know how to use as an instruction. Now, there are times and places where you will get an occasional question mark in the middle of working and valid code. This has to do with the "lister" getting out of whack on the first instruction, or it may (rarely) be a value or two a programmer has put between working code segments. But, lots of question marks are a good sign of a file.

The break or $00 command is a very enigmatic one. BRK is a very heavy debugging tool and one of the most powerful commands that the 6502 microprocessor has available. But, a break command is only rarely allowed to appear in working code as a valid instruction! Why? Because the break command immediately forces a debugging interrupt, or else, it might very rarely be used for an error trap or a program restart.

Dumb repetition is another clue. Say you push the processor status on the stack with a PHP command. That's fine. But, why on earth do it fifteen times in a row? Now, that is irrational. As you go along, you will get a feel for what is rational code and what is not.

Do it. Start through your HRCG listing. There's a few question marks at the beginning and a few breaks, but mostly it is rational code. Chances are these are stashes that go with the code modules. As you go along, you get lots of rational code. Continue some more. Page after page of rational code.

Then, suddenly, around $92DF, things get weird and stay that way, all the way to the end of the program. Lots of question marks, breaks, and really dumb code. Let's take a guess and say that our bulk file goes from $92FF to $95FF.

Now, it looks like there's some garbage, maybe a stash below $92FF, but we definitely have at least three pages of bulk file at the top.

Let's speculate. Three pages should ring a bell. Check into the HRCG Manual and you'll find it takes three pages for an alternate character set. Apparently, we have the default ASCII character set here. We absolutely should NOT jump to conclusions this early in the game, nor should we try a detailed analysis of the bulk files, but maybe just a little peek won't hurt . . . .

Check the hex dump for these pages. See the pattern? Hold it up to the light. Every eighth row almost, but not always, is all zeros. Except for the lower case g, p, and a few other exceptions, most characters would leave one dot row out of eight blank.

Strong evidence.

But, not strong enough. Later, we will tear into this bulk file and verify exactly what it does. We will also find out exactly where it starts. For now, let's draw a bright red line across the listing page between $92FD and $92FF. Label the area below this line "BULK FILE." On your serendipity pad, sketch something like Fig. 3-8, that is used to show us with an HRCG action from $8DFF through $92FE and a bulk file from $92FF through $95FF.

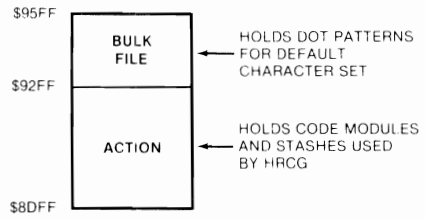


Fig. 3-8. Separating the action from the bulk files.

Don't worry, just yet, about the extra question marks we have above the bulk file. Somehow, these look "different" from the code in the bulk file. As you gain practice, these slight differences will leap out at you. But, our goal, here and

HELPFUL HINT

Use your page highliters to color the grey stripes shown on the next few pages.

Use the colors called for.

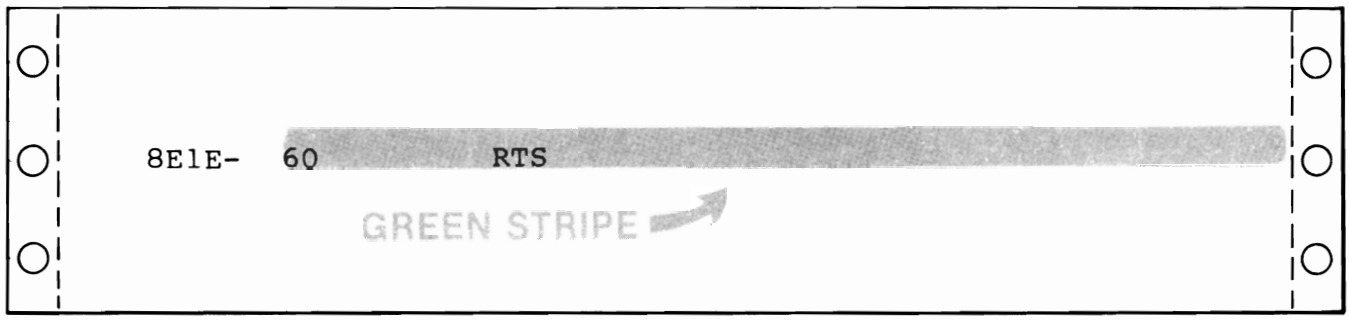
now, is only to separate the action from the bulk files, nothing more. In HRCG, this roughly cuts our task in half. In other programs, the bulk files may be the lion's share of the code.

PAINT ALL SUBROUTINE RETURNS GREEN

No matter what code you write or how secretive you are, there is an Achille's heel you have to contend with. This is the 60 RTS or *Return From Subroutine* command. RTS is our first and foremost attack point into unknown code. It is the chink in the armor, the pry point, the skeleton key. Let's split off the subroutines and watch how fast the code breaks up.

Go through your code and at every "rational" place that you find a 60 RTS, use a highliter to put a green bar through all of the code except the address.

Something like this . . .



Do this for every 60 RTS you see in the action. If you aren't sure whether the 60 is rational or not, then color *only* the RTS green, rather than the entire line. Generally, question marks *below* a 60 RTS are allowed; those close above are suspect.

If you do this on HRCG, you should end up with 35 "definites" that are greenlined all the way across, and one "maybe" located at $8F85 that is only boxed.

Do not try to analyze any of this code yet. We will let the code analyze itself later on.

We have just identified the end of every subroutine in the program. Since properly written machine-language programs will be mostly subroutines, we already have nearly all our code modules isolated! All that with several strokes of a fuzzy green page highliter!

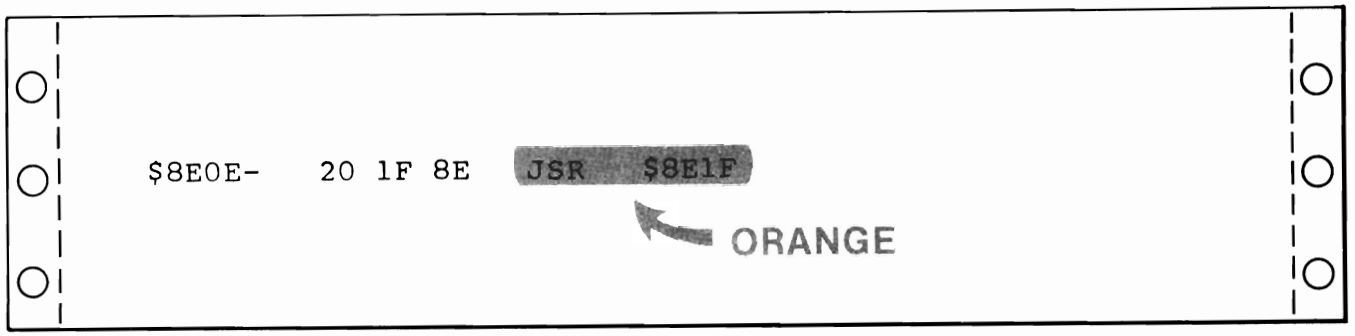
Now, things start to get interesting . . . .

PAINT ALL SUBROUTINE CALLS ORANGE

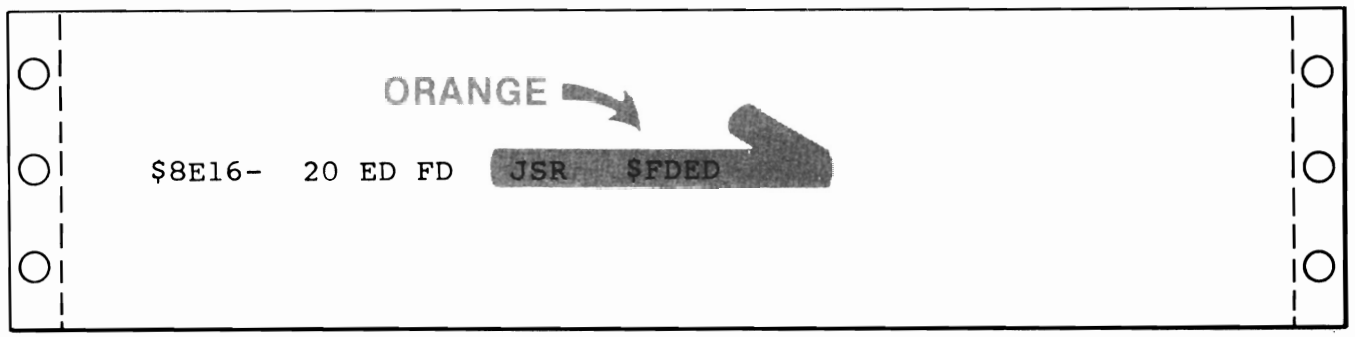
Next, get yourself an orange page highliter and go through the action. Identify every rational JSR and its address in orange.

Do this two ways. If the JSR goes to a *local* address *inside* the action, paint only the JSR and the address. If the JSR goes *out-of-range* to some *other* part of the memory, paint the JSR, the address, and one inch more, and "half" an arrowhead.

Like so for a local JSR . . .



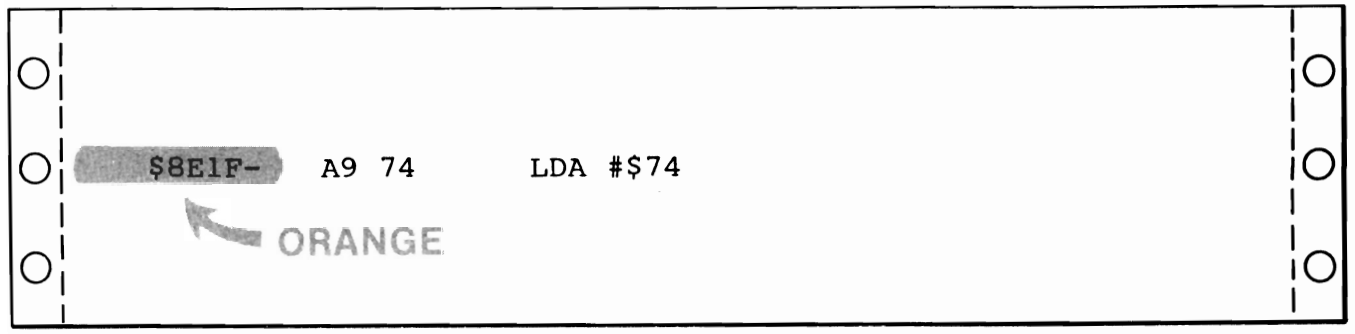
This subroutine call is in range, so we color only the JSR and the $8E1F. For an out-of-range or "long distance" call, do it like this . . .



You use the arrowhead to identify an out-of-range call. Should you have a questionable or possibly irrational subroutine call, color only the JSR mnemonic for now. (The reason for only half an arrow is that you might get two arrows side-by-side. If this happens, make one point "up" and the other "down.")

Orange is a nice color, so let's use it some more. For each and every local JSR call, find out where the JSR goes to, and color the very *start* of that line orange. Go only through the address, starting a quarter of an inch to the left.

For instance, at $8E0E, you have a local call of JSR $8E1F. Go to the start of line 8E1F and do this . . .



This tells us that we are starting on some "live" and rational code, and that what follows will be a useful and worthwhile subroutine. Once again, we do not want to analyze any code just yet.

Two fine points. If there is *already* an orange stripe here, or one of another color, just put an orange "ear" or small black dot on the existing stripe. Each new time this happens, add a new black dot.

This will give you a "popularity poll" of your subroutines. We probably won't use this voting result for our HRCG analysis, but in a large program, the popularity of a subroutine can tell you how important that sub is and how much effort you should spend in understanding it.

A second possibility is that your JSR seems to go to the middle of an op code, instead of just the start. The most likely reason for this is that the lister got off on the wrong foot. See the "WILL THE REAL LISTING PLEASE JUMP OUT" sidebar at the end of this enhancement for details on this. What happens is that the lister starts off with a value or two in a file and assumes it is a valid part of a program that can be disassembled. Op codes normally take one, two, or three bytes. If the first byte is wrong, the listing will also be wrong.

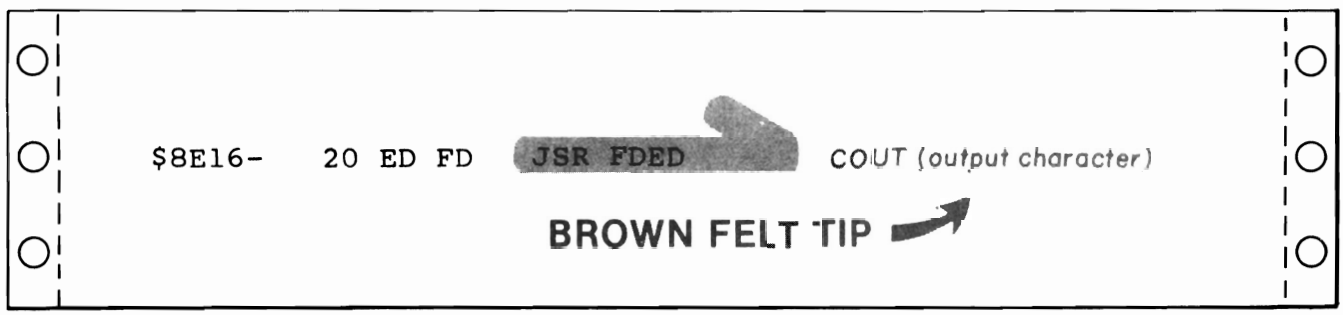
If you get a "JSR to the middle of . . .", try relisting *from the JSR address* to see if you get rational code. This will help clarify the boundaries between stashes and code modules. We will see an example of this later.

Should your JSR want to go to your bulk file, you guessed wrong! Either the bulk file has a code module in it, or else your JSR really is a random "20" in a stash somewhere. Pay careful attention to loose ends like this, for pinning down exact code and file beginning addresses can save you hours of frustration.

After all of your local subroutines are taken care of, try to identify the out-of-range ones. They *must* go somewhere. Somewhere is most often a monitor subroutine, or some DOS subroutine or subs in either BASIC.

Table 3-1 shows us the most popular locations needed by the monitor, DOS, and I/O. Try to get a match between Table 3-1 and each out-of-range subroutine call. Label this match with a brown felt-tip pen. We have purposely kept this list down to the more popular locations. We may look at Applesoft, Integer BASIC, and DOS internals in a future enhancement. Most user libraries have very extensive memory listings if you get into something out of the ordinary.

For instance, in $8E16, we have a JSR $FDED. A check of Table 3-1 shows us that it is one of the most often used monitor routines called COUT. This routine takes what is in the accumulator and outputs it as a character. This output goes to whatever is connected to the character output hooks. The code should now look like this . . .



Notice that this immediately tells us that the code module is used to output characters. This very much pins down how the module is used and its place in the big picture. And we still haven't analyzed any code.

Sometimes a JSR call will point to a different part of user RAM. This usually means that the target program is in more than one piece. Each piece, of course, will eventually have to be dealt with. The *Wizard and the Princess* is a good example of a program that has code modules all over the lot.

Table 3-1. Important Monitor, DOS, and I/O Locations

PAGE $00			
Hex	**Decimal**	**Mnemonic**	**Use**
$20	32	WNDLFT	Left side of scroll window
$21	33	WNDWTH	Width of scroll window
$22	34	WNDTOP	Top of scroll window
$23	35	WNDBTM	Bottom of scroll window
$24	36	CH	Cursor horizontal position
$25	37	CV	Cursor vertical position
$26	38	GBASL	LORES graphics base low
$27	39	GBASH	LORES graphics base high
$28	40	BASL	TEXT base address low
$29	41	BASH	TEXT base address high
$2A	42	BAS2L	Scroll temporary base low
$2B	43	BAS2H	Scroll temporary base high
$30	48	COLOR	Holds the LORES color value
$32	50	INVFLG	Normal/Inverse/Flash mask
$33	51	PROMPT	Holds prompt symbol
$34	52	YSAV	Temporary Y register hold
$36	54	CSWL	Output character hook low
$37	55	CSWH	Output character hook high
$38	56	KSWL	Input character hook low
$39	57	KSWH	Input character hook high
$45	69	ACC	Accumulator save
$46	70	XREG	X register save
$47	71	YREG	Y register save
$48	72	STATUS	Flag register save
$49	73	SPNT	Stack pointer save
$4E	78	RNDL	Keybounce random number low
$4F	79	RNDH	Keybounce random number high

PAGE $03			
Hex	**Decimal**	**Mnemonic**	**Use**
$03D0	976		Re-enter DOS
$03EA	1002		Reconnect DOS I/O hooks
$03F0	1008	BRKV	Break vector low address
$03F1	1009		Break vector high address
$03F2	1010	SOFTEV	Warm start vector low address
$03F3	1011		Warm start vector high address
$03F4	1012	PWRDUP	Warm start EOR A5 checksum
$03F5	1013	AMPERV	Applesoft "&" Jump Code
$03F8	1016	USRADR	Control Y vector Jump Code
$03FB	1019	NMI	NMI vector Jump Code
$03FE	1022	IRQLOC	Interrupt vector low address
$03FF	1023		Interrupt vector high address

Table 3-1 Cont. Important Monitor, DOS, and I/O Locations

PAGE $C0			
Hex	**Decimal**	**Mnemonic**	**Use**
$C000	−16384	IOADR	Keyboard input location
$C010	−16368	KBDSTRB	Keyboard strobe reset
$C020	−16352	TAPEOUT	Cassette data output
$C030	−16336	SPKR	Speaker click output
$C040	−16320	STROBE	Game I/O connector strobe
$C050	−16304	TXTCLR	Graphics ON soft switch
$C051	−16303	TXTSET	Text ON soft switch
$C052	−16302	MIXCLR	Full screen ON soft switch
$C053	−16301	MIXSET	Split screen ON soft switch
$C054	−16300	LOWSCR	Page ONE display soft switch
$C055	−16299	HISCR	Page TWO display soft switch
$C056	−16298	LORES	LORES ON soft switch
$C057	−16297	HIRES	HIRES ON soft switch
$C058	−16296		Annunciator 0 OFF soft switch
$C059	−16295		Annunciator 0 ON soft switch
$C05A	−16294		Annunciator 1 OFF soft switch
$C05B	−16293		Annunciator 1 ON soft switch
$C05C	−16292		Annunciator 2 OFF soft switch
$C05D	−16291		Annunciator 2 ON soft switch
$C05E	−16290		Annunciator 3 OFF soft switch
$C05F	−16289		Annunciator 3 ON soft switch
$C060	−16288	TAPEIN	Cassette tape read input
$C061	−16287	PB0	Push button 0 input
$C062	−16286	PB1	Push button 1 input
$C063	−16285	PB2	Push button 2 input
$C064	−16284	PDL0	Game Paddle 0 analog input
$C065	−16283	PDL1	Game Paddle 1 analog input
$C066	−16282	PDL2	Game Paddle 2 analog input
$C067	−16281	PDL3	Game Paddle 3 analog input
$C070	−16272	PTRIG	Reset analog paddle inputs

Table 3-1 Cont. Important Monitor, DOS, and I/O Locations

MORE PAGE $C0

Hex	Decimal	Mnemonic	Use
$C080	−16256		Disk stepper phase 0 OFF
$C081	−16255		Disk stepper phase 0 ON
$C082	−16254		Disk stepper phase 1 OFF
$C083	−16253		Disk stepper phase 1 ON
$C084	−16252		Disk stepper phase 2 OFF
$C085	−16251		Disk stepper phase 2 ON
$C086	−16250		Disk stepper phase 3 OFF
$C087	−16249		Disk stepper phase 3 ON
$C088	−16248		Disk main motor OFF
$C089	−16247		Disk main motor ON
$C08C	−16244		Disk Q6 CLEAR
$C08D	−16243		Disk Q6 SET
$C08E	−16242		Disk Q7 CLEAR
$C08F	−16241		Disk Q7 SET

Q7	Q6	ACTION
clear	clear	READ
clear	set	SENSE
set	clear	WRITE
set	set	LOAD

PAGES $F8 – $FB

Hex	Decimal	Mnemonic	Use
$F800	−2048	PLOT	Plot a block on LORES screen
$F819	−2023	HLINE	Draw a horizontal LORES line
$F828	−2008	VLINE	Draw a vertical LORES line
$F832	−1998	CLRSCR	Clear full LORES screen
$F836	−1994	CLRTOP	Clear top of LORES screen
$F847	−1977	GBASCALC	Calculate LORES base address
$F85F	−1953	NEXTCOL	Increase LORES color by three
$F864	−1948	SETCOL	Set color for LORES plotting
$F871	−1935	SCRN	Read color of LORES screen
$F941	−1727	PRNTAX	Output A then X as hex
$F948	−1720	PRBLNK	Output three spaces via hooks
$F94A	−1718	PRBL2	Output X spaces via hooks
$FA43	−1469	STEP	Single step (old ROM only!)
$FAD7	−1321	REGDSP	Display working registers
$FB1E	−1250	PREAD	Read a game paddle
$FB2F	−1233	INIT	Initialize text screen
$FB39	−1223	SETTXT	Set up text screen
$FB40	−1216	SETGR	Set up LORES screen
$FB4B	−1205	SETWND	Set text window to normal
$FBC1	−1087	BASCALC	Calculate text base address
$FBD9	−1063	BELL1	Beep speaker if ctrl G
$FBE4	−1052	BELL2	Beep speaker once
$FBF4	−1036	ADVANCE	Move text cursor right by one
$FBFD	−1027	VIDOUT	Output ASCII to screen only

Table 3-1 Cont. Important Monitor, DOS, and I/O Locations

PAGES $FC – $FD

Hex	Decimal	Mnemonic	Use
$FC10	−1008	BS	Backspace screen
$FC1A	−998	UP	Move screen cursor up one
$FC22	−990	VTAB	Vertical screen tab using CV
$FC24	−988	VTABZ	Vertical screen tab using A
$FC2C	−980	ESC1	Process escape movements A–G
$FC42	−958	CLREOP	Clear text to end of screen
$FC58	−936	HOME	Clear screen and home cursor
$FC62	−926	CR	Carriage return to screen
$FC66	−922	LF	Line feed to screen only
$FC70	−912	SCROLL	Scroll text screen up one
$FC9C	−868	CLEOL	Clear text to end of line
$FCA8	−856	WAIT	Time delay set by accumulator
$FD0C	−756	RDKEY	Get input character via hooks
$FD1B	−741	KEYIN	Read the Apple keyboard
$FD35	−715	RDCHAR	Get key and process ESC A–F
$FD62	−670	CANCEL	Cancel keyboard line entry
$FD67	−665	GETLNZ	CR, then get kbd input line
$FD6A	−662	GETLN	Get input line from keyboard
$FD6F	−657	GETLN1	Get kbd input, no prompt
$FD8B	−629	CROUT1	Clear EOL then CR via hooks
$FD8E	−626	CROUT	Output return via hooks
$FDDA	−550	PRBYTE	Output full A in hex to hooks
$FDE3	−541	PRHEX	Output low A in hex to hooks
$FDED	−531	COUT	Output character via hooks
$FDF0	−528	COUT1	Output character to screen

PAGES $FE – $FF

Hex	Decimal	Mnemonic	Use
$FE2C	−468	MOVE	Move block of memory
$FE36	−458	VERIFY	Verify block of memory
$FE5E	−418	LIST	Disassemble 20 instructions
$FE63	−413	LIST2	Disassemble A instructions
$FE80	−384	SETINV	Print inverse text on screen
$FE84	−380	SETNORM	Print normal text on screen
$FE93	−365	SETVID	Grab output hooks for screen
$FEB0	−336	XBASIC	Go to BASIC, destroying old
$FEB3	−333	BASCON	Go to BASIC, continuing old
$FEC2	−318	TRACE	Start tracing (old ROM only!)
$FECD	−307	WRITE	Save to cassette tape
$FEFD	−259	READ	Read from cassette tape
$FF2D	−211	PRERR	Print "ERR" to output hook
$FF3A	−198	BELL	Output bell via hooks
$FF3F	−193	IORESR	Restore all working registers
$FF4A	−182	IOSAVE	Save all working registers
$FF59	−167	OLDRST	Old reset entry, no autostart
$FF65	−155	MON	Enter monitor and beep spkr
$FF69	−151	MONZ	Enter monitor quietly

As you tear into your target program, go through each and every subroutine call and find out what it points to. If there are a few locations that are unexplainable, wait till later on these. Just be sure that you pin down as many subs as you can.

Now is a good time to start a separate list of which addresses go where. Label this list "Cross References" and show the *sources* of all subroutine calls. As you go along, any time that one part of the code refers to another part, add it to this list. Once again, do this *by hand,* even if you have an automatic cross-reference and disassembly program available. Eventually, you will want this list in numeric order, but for now, just list addresses as you run across them.

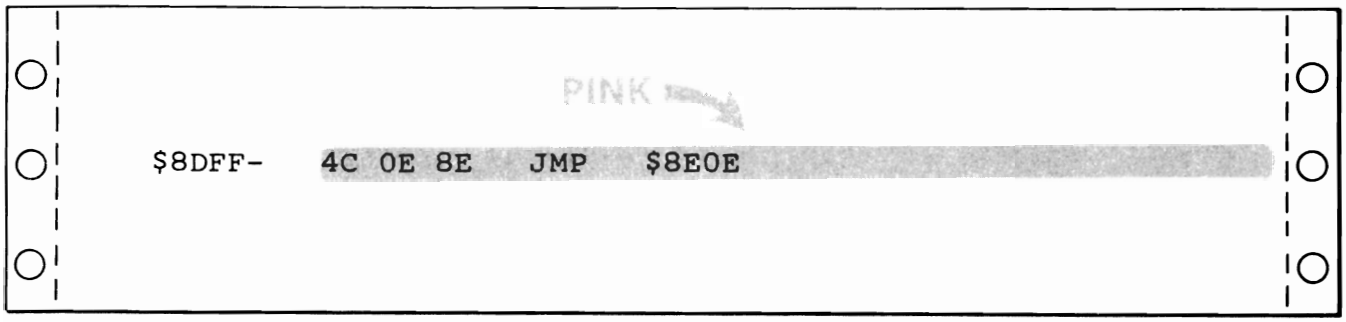
PAINT ALL ABSOLUTE JUMPS PINK

Ready for a new color? Get the pink highliter and add a pink line for any absolute JMP code ($4C) or relative JMP code ($6C). Draw the pink line all the way across the sheet for in-action jumps starting just beyond the address. Draw the pink line from the machine code to only about an inch past the operand for absolute jumps that go out of the action. End these lines with half an arrowhead like you did with the subroutine calls.

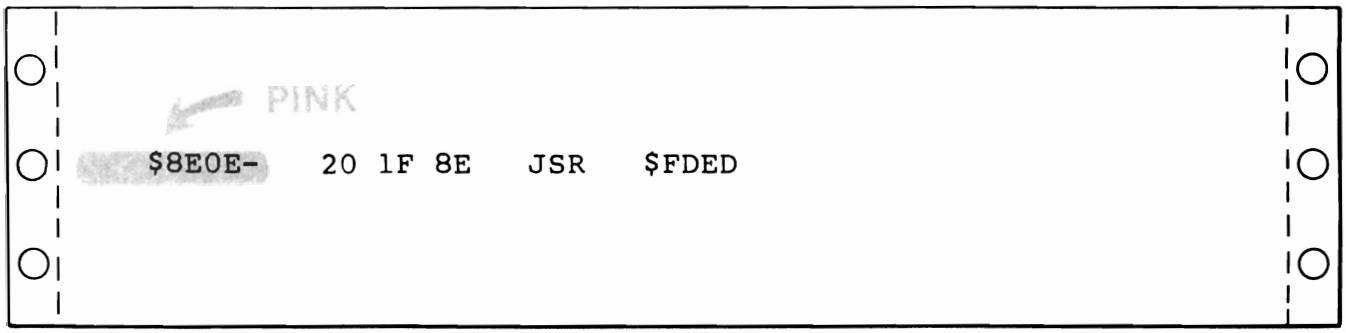
If the jumps are inside the action, then also put a pink line showing where the jump hopped to, just like you did with the subroutines. The jumper and jumpee may be connected vertically along the left-hand edge, but do this only if the two are less than twenty lines apart. Also "vote" on the most popular jumps, with dots if you see more than one jump going to a single location. Add all jumps to your cross-reference sheet.

If the jump is outside the action, use Table 3-1 to try and find out where the jump is going to. Then, label the jump using a brown felt-tip pen.

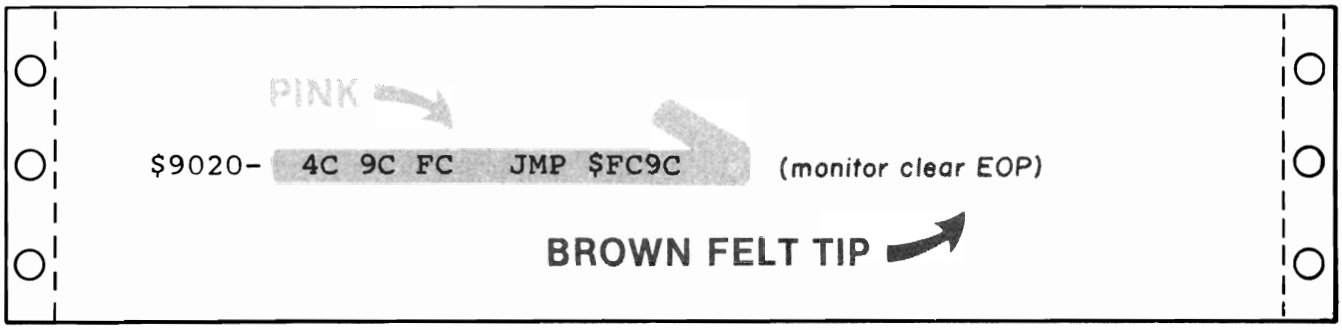
Here are the two steps that are involved in pinning down an inside-the-action jump . . .



and . . .



An outside-the-action jump looks like this . . .



Notice what is happening? The *flow* and *structure* of our program is rapidly becoming obvious. We already have all sorts of hints as to which part of the action does what. But, we are still nowhere near ready enough to tear into the code.

On an indirect jump using the ($6C) code, go to the address shown in parentheses and identify this as an indirect address, and show the location that is using it for the indirect jump.

Let's hack away at our structure some more.

SHOW THE BRANCHES IN BLUE

Get out the blue page highliter and paint each branch (BCC, BCS, BMI, BEQ, BNE, BPL, BVC, BVS, but not BIT or BRK) and its address blue. Then, go to that address and enter a blue line on the left. Finally, if the branch is less than twenty lines up or down, show the branch action with a light blue felt-tip pen. Show the direction of each branch, and keep any branch lines from crossing.

Here is an example . . .

```
9208-   C5 23       CMP   $23
920A-   90 11       BCC   $921D
920C-   2C 65 8E    BIT   $8E65
920F-   70 08       BVS   $9219
9211-   C6 25       DEC   $25
9213-   20 21 92    JSR   $9221
9216-   4C 70 FC    JMP   $FC70
9219-   A5 22       LDA   $22
921B-   85 25       STA   $25
921D-   20 24 FC    JSR   $FC24    (monitor VTAB Z)
9220-   60          RTS
```

If you find branch lines that try to cross each other, draw the problem line up the right-hand side of the address column or elsewhere as needed. It is very important to be able to glance at the listing and tell immediately which branch goes where.

We are really into our structure now. Here, the arrows jump forward, conditionally skipping part of the code. Often, the arrows will go backwards, outlining a block of code called a *loop*. The loops visually leap out at you. Check the big one at $9298. Note that there can be more than one tail connected to any given arrow.

Three refinements. The first thing is to watch out for possibly irrational code. If you are in doubt, paint only the mnemonic blue. The second is to label branches directly to RTS as an RTS, rather than showing the arrows. Finally, very long branches should show each end separately, to keep from getting too many lines on the sheets.

SEPARATE THE CODE MODULES FROM THE STASHES

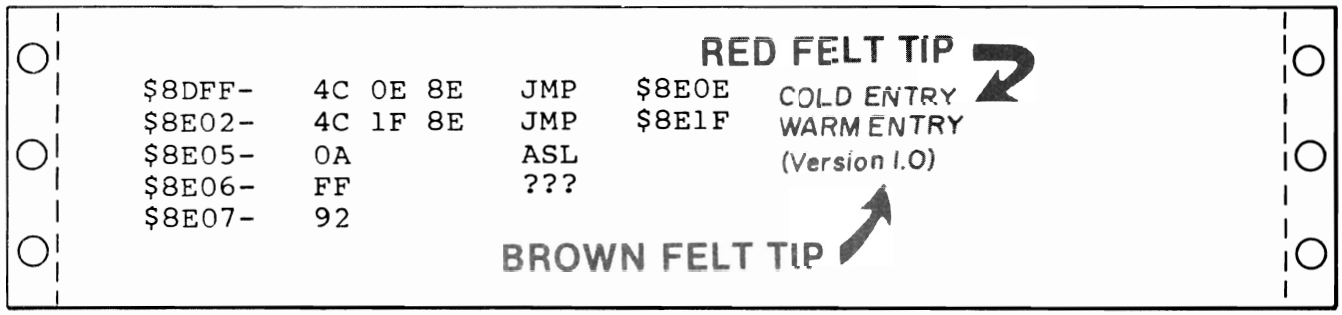
Now, carefully, look over the action and identify each "holistic" and "rational" code module. A code module should have at least one obvious entry point and at least one obvious exit point. Any question marks or lister mixups at the beginning of each module should be resolved so that we can *exactly* identify the boundary of each code module.

Then, label carefully in red all the external entry points that you know about, and any locations that the instructions refer to. Our "cold" entry point is apparently "ØR" which translates to the first code byte at $8DFF. The "warm" entry point is apparently "3R", or $8E02. The version number is at "6R", or $8E05. We see an "ØA" here, which apparently stands for version 1.Ø.

The "R" mentioned above may be new to you. The "R" means "relative" and is used with *relocatable* programs. "ØR" is the first byte in the program, regardless of where it sits; "3R" is the third byte, and so on.

By the way, if some of our example codes don't exactly fit your listing, compare the version numbers. Usually, a different version will move parts of the code up or down a few slots from where they first were.

Here's what this new stuff looks like . . .



Note that the ASL mnemonic is meaningless since we have a very short stash here holding the version number. A mnemonic is only meaningful when applied at exactly the right place in working code.

While you are labeling outside entry points, be sure to check the top of page Three for warm start, breakpoint, IRQ, NMI, and RESET vectors. These may point to important starting or recovery portions of your code. Many newer programs will RESET to themselves, rather than to the monitor. The RESET and soft start pointers can be a great help in showing you where the "high level" code sits.

Since HRCG is a utility or a service type of support program, it doesn't mess with the page Three hooks. But this is an exception, so always check.

OK. Separate your modules and identify all the external access hooks. Identify everything else that you know for certain from the program instructions.

What is left in the action consists of code modules as yet undiscovered—dead code, garbage, stashes, or oversights.

Dead code is code that is never used. Don't throw any away just yet, because it will most likely come to life later. This can happen because you have yet to discover some address entry points or else have missed coloring something along the way.

A lot of programmers will leave dead code in their programs so that the next code module or file can start off nice and neat on an even page boundary. Dead code may also be some location that will be written to later by DOS. Dead code will usually be completely rational, but it won't seem to tie in with the rest of the program.

Do not prejudge *garbage.* It may become most meaningful later on. Most programmers try to shorten their code as much as possible, so if it looks like lots of garbage is left, chances are you haven't gotten as far as you think.

Stashes are short code files that have meaning. We will attempt to identify many of them in the next section.

And *oversights,* of course, are your own doing.

We now should have identified all of the working code modules, and should be able to find most of their access and entry points, their interaction, and their exits. Now, we could actually start to think about tearing into the code.

But no, not yet. Lots of details still remain. Remember that the *longer* you hold off on finding out exactly what the code does, the *easier* the job will get, and the *less* of it you will have to do.

Let's see what the stashes and files have to say . . . .

IDENTIFY FILES AND STASHES

We have a sort of a chicken-and-egg problem. We can't tell yet what the files are up to since we don't know yet how the program works. And, since we don't know how the program works, the program can't tell us yet what the files are up to.

Fortunately, there are several *file filters* you can apply that can isolate most of the stashes and bulk files and tell you their meaning and intended use. Crack your files and you have made a tremendous progress.

Even if you can only crack a few files now, doing so is definite progress, and allows moving bytes from the unknown to the known. This is very much like a big jigsaw puzzle. Not only does each piece fit somewhere, but it also gets *removed* from the pile of unknown remaining pieces. This makes identifying and using the rest of the pieces easier since there are now less of them.

Let's isolate all the rational code modules and assume that everything left is a stash. Things may not be nearly this simple, but let's try it anyway. Fig. 3-9 shows us the remaining stash locations.

When you think you have a stash identified, put a *narrow* yellow stripe up the extreme right-hand margin, going over the tractor holes. Eventually, you want to end up with a continuous wide line up and down the right-hand side, *wide yellow* for fully known and understood stashes, and *wide green* for fully known and understood code modules. When the last of the white right margin disappears, you have conquered your target program.

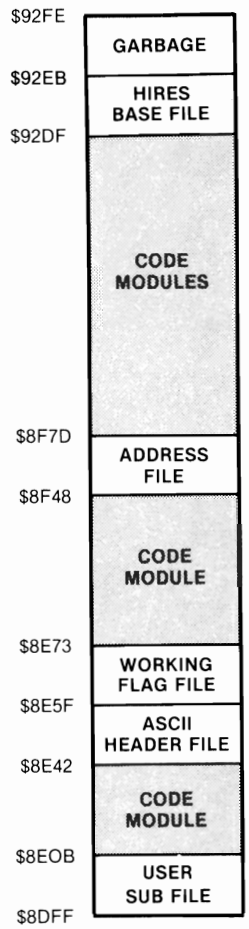


Fig. 3-9. Separating HRCG stashes from code modules.

We see a two-slot stash at $8E05, and then another obvious one starting at $8E42 on your listing. But wait. The "vibes" of our stash change dramatically at $8E5F. Let's assume we have a second stash starting there. Put a brown dotted line all the way across between $8E5C and $8E5F to remind us we think we have two separate files. The second stash apparently ends with $8E73, since $8E74 holds what looks like rational code, even though this code doesn't seem to be isolated yet.

We have a long stash starting at $8F48, obviously consisting of lots of question marks and the patently excessive use of BCC branches to dumb places. Where does this stash end? It's not obvious at first, but let's guess that it ends with $8F81. The code starting at $8F82 (that we aren't supposed to be reading yet) says to put something in $8E60 and then return. This is rational thinking, particularly since $8E60 is a slot in another stash and it might end up as a flag in a flag file.

Another stash starts at $92DF, identified by lots of zeros. Again, notice a change of vibes at $92EB. The first twelve locations are in three groups of four each and all end in zero. The remainder of the stash is strange. Let's call it two separate stashes and, once again, add a dotted brown separation line.

Now comes the tricky part. First, we want to guess what each location in each stash is used for, and, then, we want to nail each location down for sure.

To do this, make yourself up some file and stash *filters*. A stash filter is some test for some pattern that makes sense to you and to the particular target program you are attacking. The filter is valid if its answer leaps out at you and is then clinched by some independent test.

Normally, you will have to design these filters yourself. Do so very carefully. Your choice of filters will vary with the target program and how long it is. Here are some obvious filters to try first . . . .

STASH AND BULK FILE FILTERS

( ) Is it something obvious?
( ) Is it an ASCII string?
( ) Is it a table of addresses?
( ) Is it a group of flags?
( ) Is it a conversion table?
( ) Is it DOS related?
( ) Does it fill a program need?

These are the usual filters I try first and the order in which I try them. The HRCG is very accommodating in its stash uses. The early tests will tell you a lot about each stash. Other programs may not be so easy.

We attack the chicken-and-egg problem this way. First, we filter the stashes and bulk files as best as we can to find out as much about them as we are able. Then, we take this information back to the code modules and see what new thing this tells us about the modules. Then, we look into the modules and see what they tell us about the remaining unknown files.

Three or so trips round and around and we should have things pinned down fairly well. Now, if you are into an *Adventure* or something else really heavy with stashes and bulk files, it won't be this simple, but file filtering always makes a very good starting point toward further understanding.

Let's try these filters one by one and see what they tell us.

One example of an obvious file is any code on a display page. This might be $0400–$07FF for text or LORES page One, $0800–0BFF for the less common text or LORES page Two, $2000–3FFF for HIRES page One, or $4000–$5FFF for HIRES page Two. If any of these pages are in use, the bytes stored here have to correspond to the image on the screen.

Note that the screen images will change as the program is used. What you see is the code for the display pages at the *exact* point in the program where you did your listing. Chances are that text page One got messed up by the listing process itself.

Besides their obvious location, the HIRES color bytes tend to be mostly $00, $2A, $55, $7F, $80, $AA, $D5, and $FF bytes. In HRCG, we can often ignore these for a while, since they are the *result* of the program and not a part of it.

Another example of obvious code happens when you are reading interpreted BASIC statements. We'll save details on this for another time. But note that the byte patterns in BASIC are distinctive, starting with a line number, the location of the next program line, and, then, followed by a parsed code using token keywords and ASCII symbols, and, finally, ending up with an end-of-statement symbol. You can check into the LOADHRCG Applesoft program for a quick example. Do this by hex dumping machine code starting at $0800.

Usually, the BASIC code tells you that you are looking in the wrong place. But,

machine language is sometimes stuffed *inside* BASIC programs and, at other times, it will interact directly with the BASIC statements. This happens in the case of fast sort routines, variable locators, cross-reference programs, and so on.

As a much simpler and shorter example of an obvious file, look at $8E06. It is two bytes long. Is it an address? The address is $92FF. Is there anything special about $92FF? There sure is.

This is the location of the start of the bulk file that we think is an alternate character set. Since we obviously need a *pointer* like this and since a pointer would be early in the program, let's assume this stash is the pointer to the character set start.

Make sure any "obvious" evidence is very strong. Don't make wild guesses, and don't make too many guesses at once. Above all, don't force things to fit your pet theories about what a stash "has" to be. In this case, guessing an address and having that address reinforce our guess is reasonable.

Next, try some ASCII filters. The ASCII code is the standard way of stashing letters, numbers, and punctuation in your Apple. Table 3-2 shows us the ASCII code. An ASCII-coded stash will be mostly code starting with $CX or $DX, will have a few $A0 spaces thrown in, and will often end with a $8D carriage return. This assumes, as most Apple programmers do, that the ASCII most-significant bit is set to a 1. If the MSB is not set, then an ASCII file will be mostly values in the forties and fifties, with a $20 for each space, and with a $0D carriage return ending. If the file is mostly lower case, then the code will be mostly "EX" and "FX" values for a set MSB and "sixties" and "seventies" for a cleared MSB.

The actual display code used by the Apple on its upper-case-only old text screen differs slightly from ASCII. This code is shown in the Apple manual. The code provides for no control characters and offers normal, inverse, and flashing upper-case-only characters.

Programmers rarely use this *video display code* inside their programs. Instead, they usually will use ASCII, and set and clear the flashing and inverse flag (location $0032) as needed. The video display code can only be written directly to the screen and must not be output to any other device via the output hooks. The code would get used in a program only if the text display needs a wildly changing mix of flashing, inverse, and normal characters, and, then, only if the upper-case-only text screen is the only intended output.

Note that ASCII text is automatically converted to video display code by the usual monitor routines as it goes onto the screen.

Now, any file will give you *some* message back if you filter it for ASCII. The key test is whether the message says anything meaningful. You can ASCII filter all your stashes and bulk code, but it pays to pick only the most promising ones first.

In the case of the HRCG, we see that the stash beginning at $8E42 looks the most promising. ASCII filter this code and you get . . .

<dle> HI-RES CHAR GEN VERSION 1.0 <cr>

This is obviously the prompt message that first appears under HRCG. The odds of it being anything else are insanely small.

Note as you "crack" a stash, that it no longer belongs to the unknown. Further, a cracked stash will greatly simplify tearing apart the actual code, for we can now assume the code module directly above it on the listing will be involved in printing out this message.

As you get practice, you'll be able to immediately spot stashes and bulk files that will yield useful messages under ASCII code. Be sure to do this by hand a few times until you get the feel of this powerful filter.

Table 3-2. ASCII Code

upper hex digit	lower hex digit 0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0 or 8	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1 or 9	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2 or A	space	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3 or B	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4 or C	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5 or D	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6 or E	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7 or F	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

We may look at some short and powerful ASCII "snoop" programs in a future enhancement. Commercial programs that list ASCII strings can also be used. But, watch out that loading the snoop program doesn't bomb part or all of your target code. For now, do your ASCII snooping by hand till you are able to spot an ASCII file at a casual glance.

The acid test of an ASCII filter is whether you get a message back or not. Once again, don't force things. If the filter doesn't hit you over the head with the answer, try something else.

If the message seems fragmented or disjointed, possibly you are looking at an area that gets written repeatedly by DOS—putting message upon message on top of each other. A copy of the keybuffer from $0200–$02FF may also look the same way. In either case, you are far more interested in the *use* of this file, rather than its *contents.*

Our next trial stash filter should answer the question, "Do we have a list of addresses?" Look at the stash starting at $8E5F. A bunch of zeros, with a $92FF in it. Recall that 92FF points to the start of the default character generator. Do we have a file of alternate character sets here?

It doesn't look like it, but those zeros suggest a test. Let's run the HRCG and, then, let's load nine alternate character sets. Then, we will see how and if this stash changes.

Try it and there's no change! This should teach us several things. First, always be sure you have what you think you have in the machine. Second, be sure and try any trick you can think of, even if it doesn't work.

Third, and most importantly, NEVER force anything to fit your pet theories. The address filter clearly fails on this file. We'll discuss this stash in more detail later.

Let's try an address filter on the next stash starting at $8F48. Every second entry is either a $8F or a $90. Look at it on the hex dump and the addresses leap out at you. Color every second address pair yellow on your hex dump. Note that the addresses sit *backwards* on the dump, with the high byte second and the low byte first. This low-byte-first style is typical of most 6502 machine-language addresses.

It looks like we definitely have a table of addresses. For the clincher, check to see if the addresses all go somewhere rational. And, we have another surprise! Each and every address goes right in the middle of the code all right, but each one seems to point to an extremely dumb place!

Let's break our rule on tearing into code for a moment and see what code we find immediately above this address table. In this code module, we take a value, multiply it by two and, then, use it as an X index to get the high address in $8F3C. This high address gets shoved onto the stack. Then, we get the low address at $8F40 and shove it onto the stack also. Then, we return from the subroutine. What we have really done is we have faked an indirect jump to the selected module.

Now, what does a subroutine return do? It pops the stack twice and goes to the address it thought it came from. Only we just changed that with the address-low and address-high stack pushes. Note that two pushes and two pops left the stack exactly where we started. Our new code module is, therefore, at the same level that we were before, so we have done an indirect jump, rather than a JSR.

Ah! But, a subroutine return does *not* return to the address on the stack. It returns to the address on the stack *plus one!* This quirk is from the 6502 Programming Manual. Now, let's try adding one to each address and see what happens . . . .

$9046 + 1 = $9047, an immediate RTS.
$8F7D + 1 = $8F7E, a royal mess.
$8FA9 + 1 = $8FAA, the start of a subroutine!
$8FD1 + 1 = $8FD2, the start of a subroutine!
$8FEC + 1 = $8FED, the start of a subroutine!
$901A + 1 = $901B, the start of a subroutine!

etc . . .

Keep this detective work up, and we find that each address, except for the first two, points to a subroutine. The first one, an immediate return, looks as if it is a mistake.

What about the royal mess? Here is a classic example of our lister getting off on the wrong foot. Right now, the lister says . . .

```
$8F7D-  91 B0      STA   (B0),Y
$8F7F-  24 A9      BIT   $A9
$8F81-  80         ???
$8F82-  8D 60 8E   STA   $8E60
$8F85-  60         RTS
```

We know the first part of this is wrong, since we have stashed addresses and not working code here. We suspect the end of the listing may be right, since it seems rational. Our problem address is trying to point to $8F7E, so let's let it do so. Relist things starting with $8F7E, and you get . . .

```
$8F7E-  B0 24       BCS $8FA4
$8F80-  A9 80       LDA #$80
$8F82-  8D 60 BE    STA $8E60
$8F85-  60          RTS
```

And this is a nice and rational little subroutine. Our problem mess was solved by making sure our lister had something worth listing as its first entry.

Wow, what a bit of detective work. Our filter has found 27 addresses that lie in the middle of our code, all of which point to valid and workable subroutines, except for the first one that immediately returns.

Let's carry this further to see what an address stash will tell us about those subroutines. Now, 27 is one more than 26, the number of letters in the alphabet. Look at the ASCII code given back in Table 3-2, and we see a sequence of @ABCDEF . . . that jumps out at you. If the program was using a pointer that started with @ for a 00 value, we would have 27 values, the last 26 of which would be the alphabet in order. Naturally, @ wouldn't be used, so it would immediately return.

Let's take a wild guess that @ = Address 0, A = Address 1, B = Address 2, and so on. Now, let's see if this heads to any place that is useful.

Back to the HRCG manual. We have two sets of A to Z commands. This strongly suggests trying to fit the menu selections to the subroutines we already have. Right now, this is sort of a wild guess. But, if it works, and if we can prove it absolutely, we will have chopped mucho time off of our target-program attack.

Let's look further. Put a brown arrow at each subroutine's starting address that we think does something from A to Z. We get strong reinforcement right off the bat since all of them start off on a new code module.

We also notice something rather strange. Each and every module starts off with either BCS or BCC.

Odd.

But, remember that there are *two* alphabets needed in HRCG, one for the main menu selection and one for the option selection. Let's continue, since everything has been reinforced so far. Apparently each subroutine is a subroutine pair, one of which handles the "main" menu selection and one of which handles the "option" menu selection. Further, the condition of the carry flag tells us which way to go.

Which is which? To find out, we'll need more detective work.

Note that we have a function selection "E" but no option selection "E". Note also that we have an option selection "R", but no function selection "R". Go to the sixth address on the list (E is the sixth character starting with @), and we see a BCS to RTS. Apparently a *cleared* carry is a *function* and a *set* carry is an *option.*

Even more important, look at that monitor subroutine clear-to-end-of-screen leaping out at you at $9028 on your program listing. This is a solid and completely independent check on what these addresses are used for.

As a final check, we look at entry "R" (the fourteenth address), and we see a BCC to RTS, verifying that the carry flag decides which alphabet to use. The code at "R" should "Reverse the overlay" for us. A quick look at this code

shows it setting two flags for us—another confirmation. This confirmation is much weaker than the first one, but support is support.

So, go through all your code addresses and label their uses with a brown felt-tip pen. Use fairly large letters. $8F7E should be labeled "(A) function SELECT N". Check the BCS branch location and the code starting at $8FA4 gets labeled "(A) option PAGE 1 PRIMARY".

Continue through the list. The modules that use the cleared carry are functions, and the ones that use a set carry are options.

Note the power of the address filter. We now know the meaning and use of well over half of the code modules, without tearing into the code at all. HRCG is very friendly with its menu-driven selections. In other programs, you may not be able to immediately tell one address-code module from another. But the very fact that you can break up the modules into little chunks is extremely valuable and a major time saver.

The usual clue to filtering address tables is that every second entry is the same and the backwards entry pairs seem to be working through a range in a usually increasing order.

The HRCG stash at $8F48 seems to be the only table of addresses we have, so we will try some new filters.

Our next trial stash filter asks, "Do we have a group of *flags?*" A *flag* is some location that the program refers to so that it can decide what it is going to do next. In the HRCG, we can expect flags for the display page, the primary page, the working alternate character-set base address, the display mode, and so on. In an "adventure" program, the group of flags can show what is in which room, whether the giant armadillo is asleep or awake, whether the golden clockwork canary can be wound, and similar conditional things.

A flag file will often be mostly zeros, with a few FF's thrown in here and there. Other hex bytes in a flag file may have only a single bit set, such as $01, $02, $04, $08, $10, $20, $40, and $80. Flag files may also hold an occasional address or two.

One good way to verify a flag file is to find some stash that looks reasonable, and then *lightly* scan nearby code modules to find if there are references to these locations. In the HRCG, we see a likely file starting at $8E5F. A check through some of the option code shows lots of them working with locations $8E5F through $8E73.

There's usually a two-step process involved in understanding a flag file. First, you prove the flag file is there and that it is used. Then, later, when you are checking into the variables of the program in the next section, you attempt to put specific meaning onto each and every flag.

Pinning down flag meanings can be quite a challenge. The original programmer started with his flag definitions and locations and, then, built his program around them. You have to do the opposite, taking strange code and inferring what the flags originally stood for.

Our "Is it a conversion table?" filter is one that takes some experience to use. A conversion table relates addresses to data in some manner. Table lookup is a very fast way to do things, compared to calculating values. The stash starting at $92DF "looks" somewhat like a conversion table that somehow "seems" to be involved with HIRES (high resolution) base addresses. We'll keep this one a "maybe" for now.

Other examples of conversion tables are the *shape tables* and *sprite maps* used in HIRES graphics. A shape table holds a bunch of drawing directions, as needed, to directly write on the HIRES screen, using Apple's graphics routines. A sprite map will hold an image of what is to be remapped onto a HIRES screen. A character from an HRCG character-set file is an example of a sprite map.

Let's continue down the file filter list. Many machine-language programs create their own DOS, or else, use DOS variations for protection, access, and so on. In these cases, there are some *DOS filters* you can apply to your stashes. These DOS filters do not seem to help us here on the HRCG.

A file involving DOS may consist of bunches of code always ending in $X0 or $X8. These are used in the DOS nibble encoding. DOS code modules will often use header constants of $D5, $AA, $96, markers of $DE, $AA, $EB, and a trailer of $DE, $AA, and $EB. These values will jump out at you once you tune yourself into them. DOS code will also repeatedly use LDA $C08C,X commands, followed by a BNE back to itself. "X" here is the slot number. This looks real dumb when you first see it, but it is a sure sign of DOS read activity.

Another way to filter a file is to ask, "Does it fill an obvious program need?" You'll have to design suitable filters for each and every target program. Let's take a closer look at our bulk file and see what we can find out about it from its structure alone.

Visual clues can help bunches here, such as the frequency of repetition of some marker. In *Zork,* the vocabulary file has a zero and, then six bytes, over and over again. The "objects" file takes nine bytes and is in the form of seven flags and an address. Look for these patterns. Break up a file into several smaller files whenever you see any *change* in these patterns.

Even if you don't have the foggiest idea about what is in the file or how it is used, deduce as much as you can about the file structure, for this will be a great help later.

We suspect our bulk file is a default character set. All right. That means that the bits should look like characters if you arrange them just right. We know the characters are arranged in 7 × 8 squares from the ANIMATRIX program. So, a reasonable "Does it fill a program need?" filter on this bulk file is making sure to look at each and every bit and see if there is some visual pattern that looks like character dots. Let's start at $9F00 . . .

($00) — ○○○○○○○
($00) — ○○○○○○○
($00) — ○○○○○○○
($00) — ○○○○○○○
($00) — ○○○○○○○
($00) — ○○○○○○○
($00) — ○○○○○○○
($00) — ○○○○○○○

Now, that one is singularly uninformative. Yet, it is the first character and we know that the first noncontrol character in ASCII is a space. Let's try another one at $9307.

($08) — ○○○●○○○
($08) — ○○○●○○○
($08) — ○○○●○○○
($08) — ○○○●○○○
($08) — ○○○●○○○
($00) — ○○○○○○○
($08) — ○○○●○○○
($00) — ○○○○○○○

Now, that looks like an exclamation point, the second printing ASCII character. But, things are still weak. Let's try to predict a quote for the next one, starting at $930F. And, sure enough . . .

($14) — ○○●○●○○
($14) — ○○●○●○○
($14) — ○○●○●○○
($00) — ○○○○○○○
($00) — ○○○○○○○
($00) — ○○○○○○○
($00) — ○○○○○○○
($00) — ○○○○○○○

Apparently the characters are in the character file in order, just like they go on the screen. Only, we may be jumping to conclusions. Let's try several more characters. There are 96 characters, each of which takes up 8 bytes, so we can expect 768 bytes total, or exactly 3 pages. Thus, we would expect the numbers and punctuation to start at $92FF, the upper-case alphabet at $93FF, and the lower-case alphabet at $94FF.

To prove this, we would expect a capital "A" to be at $93FF + $08 = $9407. Try it, and lo and behold . . .

($08) — ○○○●○○○
($14) — ○○●○●○○
($22) — ○●○○○●○
($22) — ○●○○○●○
($3E) — ○●●●●●○
($22) — ○●○○○●○
($22) — ○●○○○●○
($00) — ○○○○○○○

So, obviously, we know everything that we should know about the bulk file now, right?

Wrong!

One very important rule . . . .

No matter where you are in cracking a file, there is ALWAYS one surprise remaining between where you think you are and where you really are.

THE FINAL SURPRISE IS THAT THERE ARE NO MORE SURPRISES!

Always, check things as independently and as completely as you can before convincing yourself that something is so. In the case of our bulk file, the surprise comes on the next character.

($1E) — ○○●●●○
($22) — ○●○○○●○
($22) — ○●○○○●○
($1E) — ○○●●●○
($22) — ○●○○○●○
($22) — ○●○○○●○
($1E) — ○○●●●○
($00) — ○○○○○○○

Uh — whoops. That's a B all right, but why is it backwards? All the rest are obviously frontwards, aren't they? Let's try the next character . . .

($1C) — ○○●●●○○
($22) — ○●○○○●○
($02) — ○○○○○●○
($02) — ○○○○○●○
($02) — ○○○○○●○
($22) — ○●○○○●○
($1C) — ○○●●●○○
($00) — ○○○○○○○

Hmmm . . . , the "C" is also backwards. But why would some characters be frontwards and some backwards?

They wouldn't.

The first three characters that we looked at just happen to look the same frontwards or backwards. That's the prize we find in this particular box of *Crackerjacks.*

Apparently all of the characters are "backwards" with the least-significant bit going out to the display first and the most-significant bit going out to the display last. Think about this for a while and you'll remember that a backwards entry is also how all of the HIRES color routines work, so we should have expected something like this.

Fig. 3-10 shows us the final arrangement of the default character set in the bulk file. We can safely assume that all other character sets will behave the same way, even though they are located elsewhere in memory.

Now, a visual bit-by-bit check of a long file may turn out to be totally worthless. But, it also may be a sure clue that will permit quickly cracking most of the program code. It all depends on the program and how creative your cracking approach is. What you have to do is make up a "Does it fill a program need?" filter that might show you something. But, keep trying things that are geared to the target program until something leaps out at you and hits you over the head.

There is one ultimate file filter . . . .

THE ULTIMATE FILE FILTER

Fill the file with water and see where it leaks.

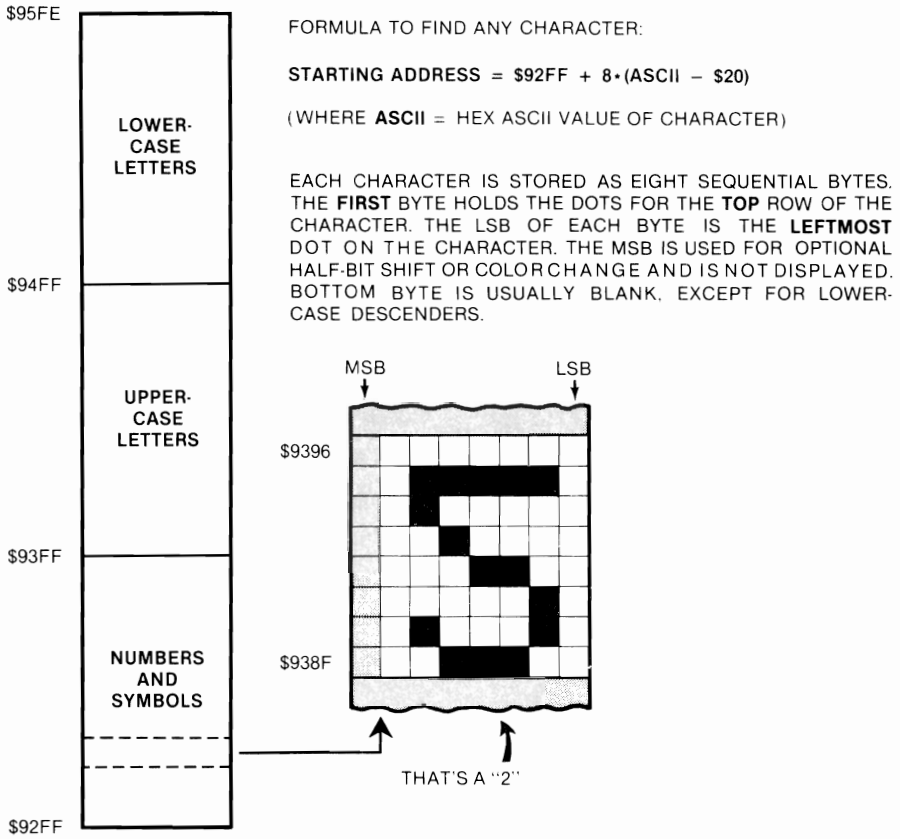


Fig. 3-10. How the HRCG default character set is stored in the bulk file.

If all else fails, and you are making reasonable progress elsewhere in your attack, try changing some or all of the contents of a file and see what *changes* take place in the program.

Usually, the program will bomb on random file changes. But, by finding out *where* and *when* it bombs and, then, zeroing into one or two locations in our target file, we can sometimes find out lots of things in a hurry.

Suppose we didn't know our bulk file was an alternate character set. If you made the first eight bytes all $FF's instead of $00's, then all the spaces in any message would be white boxes, but nothing else would change. Now, this would immediately tell you that the file was a character set and that the first entry was a space.

Another neat example of this is to go through the movable object file in an Adam's Adventure and change all the room numbers to $FF. You are now carrying . . . . . everything!

The only unexplained file left in HRCG is the stash starting at $92EB. Now, this code seems downright weird and has failed all the other tests. The code could be garbage since it is at the very end and since the character generator sets all have to start at the same base address.

Fill this file with $FF's and what happens?

Nothing.

There is no change in any part of HRCG that is immediately obvious. So call it garbage.

At this point, you should have all your stashes and all your bulk files separated and many of them fully identified.

Back to the code modules . . . .

ATTACK VARIABLES AND CONSTANTS

Start a fresh page on your quadrille pad and head it "LIST OF VARIABLES."

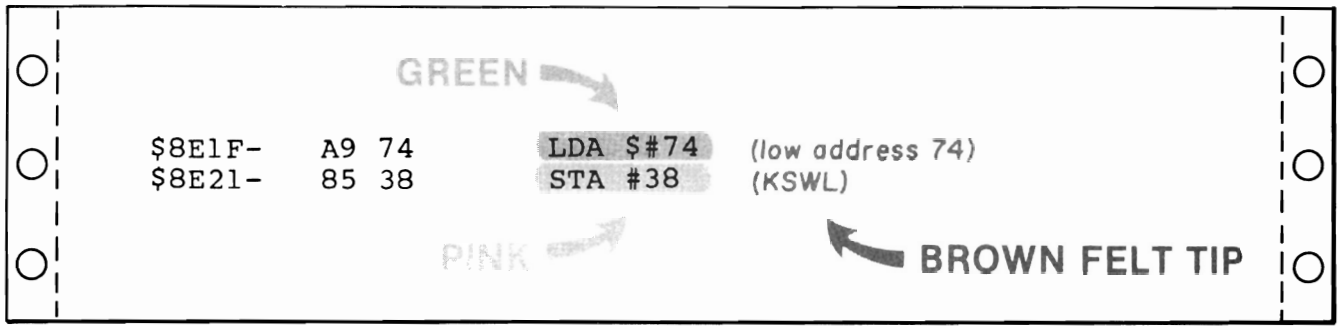
Now, go through the code modules line by line, and each time you find an address used for loading, storing, BIT testing, logic operations, or whatever, paint the variables pink and the constants green.

Note that the constants will always have a # symbol in front of them. Page Zero addresses will be two hex digits but no #. Absolute addresses will be four hex digits, again with no #.

As an example, an LDA $05 puts what is in page Zero memory location $0005 into the accumulator. This is a variable. It is a variable since the contents of $0005 can have any of 256 values ranging from $00 through $FF. But, an LDA #$05 puts the *value* hexadecimal $05 into the accumulator. This is a constant equal to "five" of something.

Watch for that # symbol! It will get you every time if you ignore it.

Our first code module starts at $8E1E. Your variable and constant lines should look like this . . .



As you identify variables and constants, you can start tearing into code. But, if something isn't immediately obvious, go on elsewhere. Our first object here is getting a list of all locations that get used for target-program variables. However, if we can find the meanings at the same time, we are just that much further ahead.

The code starting at $8E1F is very easy to read. First, we set the input hook to $8E74 and, then, we set the output hook to $8F18. Next, we reconnect DOS to internalize these hooks. Then, we switch to the full graphics and pick the HIRES mode. Continuing, we restore the default display parameters and, then, we switch on the graphics mode. Finally, we exit.

How did we figure all that out? Look back at what we know about these variables. . . .

FROM TABLE 3-1 OR PREVIOUS RESULTS

$38 and $39 . . . . . are the KSW switches in the monitor.
$36 and $37 . . . . . are the CSW switches in the monitor.
$03EA . . . . . is the DOS reconnect hook.
$C052 . . . . . is the full screen switch.
$C057. . . . . is the HIRES switch.
Sub $9158 . . . . . is named "Restore Default Parameters."
Sub $900D . . . . . is named "Display Primary."
$C050 . . . . . is the GRAPHICS switch.

Usually, you won't be so lucky on your first try. We now understand that this code module is the initialize portion of HRCG. We also now see what it does. We add all the above variables to our variables list, and color everything that we understand reasonably well pink for a variable or green for a constant.

Since this module is so obvious, we can also color the right tractor margin a solid wide green.

We also found out something new. All keyboard inputs go to $8E14 and all character outputs go to $8F18. So, label these locations in red. Do this and two more large code modules now have labels. Call these KEYBOARD ENTRY and CHARACTER OUTPUT.

Continue through the code modules and identify every variable. If you can tell exactly what the variable is used for, so much the better. If not, just put the variable on the list. The variable will most likely crop up later in another code module that may clarify its use.

Don't go overboard on analyzing code. If something is obvious and simple, go ahead and crack the code. If it is not, just record all the variables. Do not color any variable or constant till you understand what it is used for. But, be sure to get all of them on the list.

Pay particular attention to variables inside parentheses. A set of parentheses means that you are doing a jump indirect or using one of the indexed indirect

Table 3-3. List of Variables for HRCG

ADDRESS	MNEMONIC	USE
	— page $00 —	
$0020	WNDLFT	Left end of scrolling window
$0021	WNDWDTH	Width of scrolling window
$0022	WNDTOP	Top of scrolling window
$0023	WNDBTM	Bottom of scrolling window
$0024	CH	Text screen cursor horizontal
$0025	CV	Text screen cursor vertical
$0028	BASL	Text screen base address low
$0029	BASH	Text screen base address high
$002A	BAS2L	Dot row HIRES base address low
$002B	BAS2H	Dot row HIRES base address high
$0035	YSAV1	Temporary Y register save
$0036	CSWL	Character output hook low
$0037	CSWH	Character output hook high
$0038	KSWL	Keyboard input hook low
$0039	KSWH	Keyboard input hook high
$004E	RNDL	Keyboard delay low
$004F	RNDH	Keyboard delay high
$00EB		Temporary X register save
$00EC		HIRES base address low
$00ED		HIRES base address high
$00EE		Character set base address low
$00EF		Character set base address high
$00FF		Temporary accumulator save
	— page $01 —	
$0104		JSR stack source pointer (,X)
	— page $03 —	
$03EA		Hook to reconnect DOS

Table 3-3 Cont. List of Variables for HRCG

ADDRESS	MNEMONIC	USE
	— page $8E —	
$8E06		Default character set base low
$8E07		Default character set base high
$8E08		Jump to user sub A
$8E0B		Jump to user sub B
$8E42		Pointer to header message
$8E5F		Escape key flag
$8E60		Alternate character set flag
$8E61		Primary page flag
$8E62		Inverse video flag
$8E63		Transparent video flag No. 1
$8E64		Transparent video flag No. 2
$8E65		Scrolling flag
$8E66		Case flag
$8E67		Character set in use base low
$8E68		Character set in use base high
$8E69		Save of $8E61 while block mode
$8E6A		Save of $8E62 while block mode
$8E6B		Save of $8E63 while block mode
$8E6C		Save of $8E64 while block mode
$8E6D		Save of $8E65 while block mode
$8E6E		Save of $8E66 while block mode
$8E6F		Save of $8E67 while block mode
$8E70		Save of $8E68 while block mode
$8E71		Block mode flag
$8E72	CH	Horizontal cursor position
$8E73	CV	Vertical cursor position
$8F48		Function address file base low
$8F49		Function address file base high
	— page $92 —	
$92DF		Start of HIRES pointer file
$92FF		Default character file start
	— page $C0 —	
$C000	IOADR	Keyboard ASCII input
$C010	KBDSTRB	Keyboard strobe reset
$C052	MIXCLR	Full graphics soft switch
$C054	LOWSCR	Page 1 soft switch
$C055	HISCR	Page 2 soft switch
$C057	HIRES	HIRES soft switch
$C050	TXTCLR	Graphics soft switch
	— page $FC —	
$FC22	VTAB	Vertical tab from CV sub
$FC24	VTABZ	Vertical tab from accumulator
$FC42	CLEEOP	Clear to end of page sub
$FC58	HOME	Home text screen monitor sub
$FC70	SCROLL	Scroll text monitor sub
$FC9C	CLREOL	Clear to end of line sub

modes. These are among the most powerful commands the 6502 microprocessor has available, so it pays to very carefully understand how these are used. It really gets challenging when you get into the double or even triple indirect file manipulations that are involved in the longer *Adventure* programs.

Don't worry too much about fuzziness and loose ends. Identify what you can and crack what code you can, but *keep moving!* And, every time you get a new piece of checkable information, go back and plug it in everywhere it seems to fit. The ripple effect when you do this is often astounding.

Our flag file bytes get identified as you go along. Note that $8FA4 puts a $20 in 8E61 to display the primary page and that $8FCC puts a $40 in $8E61 to display the secondary page. We can then conclude that $8E61 is the page flag. You can continue this reasoning for the other flags. The block mode ends up using the bottom half of the flag file.

You should end up with a complete list of all variables, some of the code completely cracked, and lots of new hints that will help you elsewhere in your attack.

After your list is nearly complete, recopy it legibly in numeric order. Table 3-3 shows a list of the variables used in HRCG. Use this as an example.

PAINT THE HOUSEKEEPING YELLOW

Next, go back through the code. Every code line that uses an *implied* addressing mode should be painted yellow once you understand it. Implied mode instructions use a single op code byte and are not qualified by a value or an address. Examples are INX, DEY, TXA, CLD, SEC, TSX, and so on.

If you happen to have code that uses the stack to hold a value for you, this will show up with a PHA, some operations, and, then, a restoring PLA. Show these in yellow just like any other implied instruction. But if, and only if, the PHA and PLA are irrevocably paired as a temporary store, connect them with a yellow bracket.

Like this . . .

```
8E87-   B1 2A       LDA   ($2A),Y
8E89-   48          PHA
8E8A-   E6 4E       INC   $4E
8E8C-   D0 0B       BNE   $8E99
8E8E-   E6 4F       INC   $4F
8E90-   CA          DEX
8E91-   D0 06       BNE   $8E99
8E93-   49 7F       EOR   #$7F
8E95-   91 2A       STA   ($2A),Y
8E97-   A2 50       LDX   #$50
8E99-   2C 00 C0    BIT   $C000
8E9C-   10 EC       BPL   $8E8A
8E9E-   68          PLA
8E9F-   91 2A       STA   ($2A),Y
8EA1-   BA          TSX
```

Once you understand how a yellow line is used, add comments in brown to explain it. Should you get paired PHP and PLP commands, these should also get bracketed in yellow, but only if they always work together.

What you are after here is to have a color on each and every line, a comment on each and every line, and, on the right margin of the page, a solid green area for each module that is understood, and a solid yellow area for each stash that is cracked.

WRITE A SCRIPT

Where you are right now depends on your experience and how tough and how long the program is. If you try this method on a target program that is only a few hundred words long, you should be done by now. You should not only have met your limited goal, but should have the rest of the entire program completely cracked. On longer programs, the chances are there is lots of white space remaining. These white spaces point to uncracked code and unbroken stashes and bulk files.

The next step is to write a *script.* Explain in people-type words what each and every known stash, bulk file, and code module does.

A complete script of HRCG appears in Table 3-4. Use this as an example. If you have to leave blanks for now, do so.

CUSTOMIZE YOUR ATTACK

Hopefully, you will know what to do next at this point. Go on your own vibes in the most obvious direction.

Obviously, all machine-language programs are different. Some will involve themselves a lot with DOS. Others will use only the HIRES screens for game actions. Still others will interact with a host BASIC program, and so on.

What you now want to do is customize the attack to fit the program. How you do this is up to you. Here are some things I sometimes try . . .

CUSTOM ATTACK METHODS

( ) Look for built-in diagnostics.
( ) Use breakpoints.
( ) Try flowcharting.
( ) Attack indirect addressing.

( ) Add hooks.
( ) Gain partial control.
( ) Use the cassette.
( ) Single step and trace.

( ) Chip away at it.
( ) Attack the fundamental subs.
( ) Ask for help.
( ) Use partial boots.

( ) Detect changes.
( ) Alter files.
( ) Put program on an assembler.

( ) Attack a similar program.
( ) Decipher special codes.
( ) Try something easier.

That's sure a long list. Not every idea will work on every program, though. Let's look at a few of these in more detail, after a page or two.

Table 3-4. Complete Script of HRCG

ADDRESS		COMMENTS
$8DFF	-	Hard entry point. Clears screen and prints header, connects HRCG hooks.
$8E02	-	Soft entry point. Connects HRCG but does not clear screen.
$8E05	-	Version number × 10.
$8E06–8E07	-	Base address of default character-generator set. Defaults to $92FF.
$8E08–8E09	-	User subroutine A starting address called by option Y. Defaults to subroutine return RTS.
$8E0B–8E0C	-	User subroutine B starting address called by option Z. Defaults to subroutine return RTS.
$8E08–8E1E	-	Hard entry routine. Sets I/O hooks, then reconnects DOS. Switches to HIRES full screen. Restores DOS and default parameters. Displays primary. Switches to graphics.
$8E42–8E5C	-	Stash holding ASCII-coded title and version. Used during cold entry.
$8E42–8E73	-	Stash holding all working flags —
		$8E5F - $80 if previous key ESC $00 otherwise
		$8E60 - $80 if alternate characters $00 if default characters
		$8E61 - $20 if page 1 primary $40 if page 2 primary
		$8E62 - $00 if normal video $7F if inverse video $80 if overstrike video $C0 if complement video
		$8E63 - $80 if transparent mode $00 otherwise
		$8E64 - $60 if transparent mode $00 otherwise
		$8E65 - $00 if scrolling $FF if wraparound
		$8E66 - $00 if caps lock - $80 if lower case - $C0 if single capital
		$8E67 - Base add low of set in use
		$8E68 - Base add high of set in use
		$8E69 - Save of $8E61 while block
		$8E6A - Save of $8E62 while block
		$8E6B - Save of $8E63 while block
		$8E6C - Save of $8E64 while block
		$8E6D - Save of $8E65 while block
		$8E6E - Save of $8E66 while block
		$8E6F - Save of $8E67 while block
		$8E70 - Save of $8E68 while block

Table 3-4 Cont. Complete Script of HRCG

ADDRESS		COMMENTS
		$8E71 - $00 if normal display $FF if in block mode $8E72 - CH horizontal position $8F73 - CV vertical position
$8E74–8EAC	-	Enter HRCG via keyboard hook. Save A, X, BASH, and BASL. Debounce keyboard and flash cursor till key is pressed. Reset keyboard strobe.
$8EAD–8F17	-	Check keyboard for ESC or CR. If a CR, process via sub $928D. If an ESC, process I, J, K, M for cursor motions. Then, clear EOL if E or clear EOS if F. Process A, B, C, and D cursor motions.
$8F18–8F27	-	Enter HRCG via output hook. Save A, X, and Y. If a number and preceded by ESC, change character-set number via $8F86. If a control command, clear Carry if a function and set Carry if an option. If a letter from @ to Z, process by getting address from stash $8F48 and doing an indirect jump.
$8F48–8F7D	-	Stash of 27 addresses for menu selections A–F. Selection @ does an immediate RTS. Address picked by $8F28.
$8F86–8FA3	-	Function A. Alternate character set. If a number from 0–9, calculate new base address and store in $8E67.
$8FA4–8FA9	-	Option A. Put #$20 in flag $8E61 to switch to primary page 1.
$8FAA–8FCB	-	Function B. Begin block display if not already there. Put $#FF into flag $8E71. Move flags $8E61 through $8E67 to $8E69 through $8E70 as temporary save. Move CV and CH into flags $8E72 and $8E73.
$8FCC–8FD1	-	Option B. Put #$40 in flag $8E61 to switch to primary page 2.
$8FD2–8FDE	-	Function C. Carriage return. If not below bottom, do CR via $9204.
$8FE2–8FEC	-	Option C. Complement display by making flag $8E63 a #$C0 and $8E64 a #$00.
$8FED–900C	-	Function D. Block display off. If in block mode, move flags back to $8E61–8E68. Reset block flag and CH flag to zero, CV flag to bottom.
$900D–901A	-	Option D. Display primary. Switch to page One. Check primary flag and switch to primary flag page.
$901B–9022	-	Function E. Clear HIRES page to EOL using $928D. Then, clear text page using monitor CLEOL.

Table 3-4 Cont. Complete Script of HRCG

ADDRESS		COMMENTS
$9023–902A	-	Function F. Clear HIRES page to EOS using $927A. Then, clear text page using monitor CLEOS.
$902B–903F	-	Function H. Backspace. Go left one character if entry at $902B. If screen left, go up one line.
$9040–9047	-	Function I. Set inverse video flag by putting #$75 into $8E62.
$9048–904F	-	Function K. Set caps lock flag by putting #$00 into $8E66.
$9050–9057	-	Function L. Set lower-case flag by putting #$80 into $8E66.
$9058–9072	-	Unsupported function M. Apparently a scroll diagnostic, once reached by CTRL-S, CTRL-C.
$9073–907A	-	Function N. Set normal video flag by putting #$00 into 8E62.
$907B–9082	-	Function O. Set option flag by putting #$40 into $8E60. Next key will complete option command.
$9083–908D	-	Option O. Pick overstrike mode by #$00 into 8E63 and #$00 into 8E64.
$908E–9095	-	Function P. Clear HIRES page via $9270 and text page via monitor HOME. Note that an image of the HIRES screen is put on text page 1.
$9096–909E	-	Option P. Pick print mode by putting #$00 into $8E63 and $8E64.
$909F–90AD	-	Function Q. Home cursor inside text window. Move upper-left values to CH and CV. Then, reset text screen via monitor VTAB.
$90AE–90BA	-	Function R. Reverse overlay by putting #$C0 into $8E63 and #$60 into $8E64.
$90BB–90C2	-	Function S. Shift next character by putting #$C0 into flag $8E66.
$90C2–90C8	-	Option S. Pick scroll mode by putting #$00 into flag $8E65.
$90C9–90D5	-	Option T. Set transparent mode by putting #$80 into $8E63 and #$60 into $8E64.
$90D6–9103	-	Function V. Text window, upper left, by resetting WNDLFT and WNDTOP after check for on-screen values. Transfers vertical position to CV flag if not in block mode.
$9104–9124	-	Function W. Text window, lower right, by resetting WNDWDTH and WNDBTM after check for on-screen values.

Table 3-4 Cont. Complete Script of HRCG

ADDRESS		COMMENTS
$9125–912A	-	Option W. Set wrap mode by putting #$FF into $8F65.
$912B-914E	-	Function Y. Open to full text screen by putting #$00 into WNDLFT and WNDTOP and #$28 into WNDWDTH and $#18 into WNDBOTM. Save as CH and CV flags if not block mode.
$914F–9151	-	Option Y. Call user subroutine A by jumping to jump command stored at $8E08. Defaults to RTS.
$9152–9177	-	Function Z. Restore defaults. Reset all flags to #00. Set full text window. Pick default character set. Display primary page. Reset user subs to RTS.
$9178–917A	-	Option Z. Call user subroutine B by jumping to jump command stored at $8E0B. Defaults to RTS.
$917B–9196	-	Begin character entry. Exit RTS if option flag set. Check case mode and change to upper case or reset shift flag if needed.
$9197–91C4	-	Continue character entry. Calculate character location and save as $EE and $EF. Calculate screen base address location and save as $EC and $ED. This is the top dot row for any character position. The running dot row address gets held in $2A and $2B. Then, the character is saved on page One text screen. Like so . . . $28–29 - text screen base address $2A–2B - HIRES dot row address $EC–ED - HIRES base address $EE–EF - Character-set base address
$91C5–91F7	-	Continue character entry. For eight dot rows, get the character dots and inverse if needed. Get the dots already on the screen; then, AND or OR with character dots if needed. Then, return result to the screen. Next, calculate the address of the next lower dot row and repeat till all of the characters have been entered.
$91F8–9220	-	Move cursor. Go one to the right unless at extreme right of the window. If a CR is needed, go down one line unless at extreme bottom of window. If at bottom, check flag for scroll or wraparound, and continue.
$9219–9220	-	Wraparound mode. Set WNDTOP to top of text window. Do a monitor VTABZ to recalculate base addresses.
$9221–926F	-	Scrolling mode. Dot line source is $2A–$2B. Destination is address

Table 3-4 Cont. Complete Script of HRCG

ADDRESS		COMMENTS
		$EC–$ED. Destination is eight dots above source. Starting at the top of the screen, scroll downward, loading from ($2A) and storing at ($EC). Y register handles CH position, stepping from WNDWDTH downward. X register handles position of eight rows per character. One entire dot row is entered, then another until done. After a line is remapped, the base address of the next line is calculated, making the old source the new destination, and calculating a new source. This continues until the entire screen is mapped. The bottom line is then cleared via $8E63.
$9270–9279		Clear screen. Set CV to WNDTOP and CH to WNDLFT and continue via $927E.
$927A–928C		Clear to end of screen. From present CH and CV, clear to EOL via $9291 as often as needed to empty screen.
$928D–92CA		Clear to end of line. For eight dot rows, calculate address, then remove character from screen. Inverse background if needed. Y register works from CH to WNDWDTH doing one dot row at a time. X register handles dot rows, working from top of character down.
$92CB–92DF	-	Calculate HIRES base address. Divide CV by two. Go into the table in $92DF–92EA and lookup base address value. Process this value and store in $2A and $2B.
$92DF–92EA	-	A stash of table lookup values used to calculate HIRES base addresses needed by $92CB or $92CD.
$92EB–92FE	-	Apparently unused garbage.
$92FF–96FE	-	Bulk file of default character set. Holds dot patterns of all ASCII characters. The seven least significant bits hold the horizontal dot pattern IN REVERSE for one dot line. Eight successive bytes hold the dot pattern for one character, arranged from top to bottom. Locations $92FF–93FE hold numbers and symbols. $93FF–94FE hold upper-case alphabet, and $95FF–96FE hold lower-case alphabet. Bottom dot row is blank except for descenders. 96 characters total.

If you are attacking a very complicated target program, chances are the original author may have had some of the very same problems you did. And,

if he was smart enough, he just, possibly, may have built in some problem-solving diagnostics.

For instance, the *Adam* adventures have a "Possible" and a "Did" tracing debugger that you can access with two keystrokes. *Zork* includes a hook that lets you stop the action after each code module, and print out whatever you like, such as the files just accessed. *Zork* will also give you a complete list of rooms with just a few keystrokes. A few minor changes to *Wizard and the Princess* and you get a guided picture tour of all the rooms.

Be on the lookout for any diagnostic helps that may be built into the program. Then, see just how you can tap them.

Breakpoints are another way to tackle a program. What you do is reach into the target program at a place where you want it to stop, and insert a $00 or BRK command. When the Apple reaches this point in the program, it will stop and immediately do a software interrupt.

What happens next is decided by which monitor ROM you have in use. If you have the old ROM, the break puts you in the monitor and displays all of the working registers. If you have the autostart ROM, the BRK command does a jump indirect to the address contained in locations $03F0 (low) and $03F1 (high). You can go from this address into the monitor, or else, directly to another snoop program that spells out what each and every pointer and indirect address is up to.

There is one clinker in the works when you use BRK. You might need the old ROM to gain control of the program so that you can change $03F0 and $03F1, and then switch to the autostart one. A "protected" program under autostart will never let you get down into the monitor or change any locations. Use of either ROM card with a hardware change-over switch often can get you out of this bind.

A breakpoint can be used as anything from a scalpel to a cannon, depending on what you want to do and how large a hole you want to blast in the target program.

Drawing a flowchart may help you. I don't use that method too much since it sounds like something the dino people would want you to do.

The addressing modes that give the 6502 microprocessor its extreme power are the indirect ones. These include jump indirect, indirect indexed, and the rarely used indexed indirect. All of these are identified by an address in parentheses following the mnemonic. A lot of setting up is needed to use these locations. Most often, an address pair on page Zero has to be set up ahead of time.

Understanding the *real* address used for an indirect instruction can be the key to cracking tough codes. It pays to spend lots of time being sure you know exactly where these addresses are going to and the reasons that they are doing so.

Things really get interesting when you get involved in double and triple indirect addressing, as is common in adventure programs. The code may go to some base address, pick an address pair out of a file there, and use that address as an indirect pointer in another instruction. If the files happen to be longer than 256 bytes, then double indirect is needed, rather than a simple indexed instruction.

Patience and practice are essential to cracking indirect codes. If all else fails, replace the indirect op code with a BRK command. On the break, get into the monitor and check the locations used to hold the indirect address.

Hooks are attachments you make to the program to gain partial control. You might write your own small "host" program and let it "borrow" subroutines off the target program. This is one possible way to dump files off protected disk

tracks. Once you are able to use and control key subroutines in the target program, you are well on your way to solving everything else.

The tape cassette is often ignored. Yet the tape system is a very valuable tool. One "protection" scheme used involves putting a program in the same space where Apple DOS 3.3 would normally reside. A custom DOS is then put somewhere else and there is no immediate way to save the program entered under DOS 3.3, since booting the DOS 3.3 overwrites and, thus, destroys the program.

But, the cassette doesn't care. It can save any code in any location at any time. One thing you can do is move the target code down in memory below DOS, save it to cassette, and then boot the DOS. Save this lower version on DOS and, then, add a "move" command that puts it back where it wants to sit.

Cassettes are also useful in upgrading between various DOS versions. They are slow, unreliable, and unwieldy, but they just might work if all else fails.

The single-step and trace features on the old monitor are very useful on some parts of some programs, particularly if you dump them to a printer. But watch out that you don't try to trace a delay loop, such as the one that waits for a disk drive motor to come up to speed. The trace operation slows things down some 10,000 times from normal speed, so a two-second delay will take several days and miles of paper to print. Sometimes you can break into the loop, reset the counter locations, and continue. Other times, you'll have to combine single step or trace with breakpoints. Run the code till you hit the breakpoint, and then single step from there.

Tracing to a printer is one very good way to crack indirect addresses to find the files that they work with.

Beware of tracing parts of programs that read the screen, since tracing and displaying can interact. For instance, a clear-to-end-of-screen will hang during a trace, since trace keeps resetting the screen locations. If you are printing, defeat the screen echo during these times.

Another custom attack method is to chip away at the target. Your goal may seem to be hopelessly buried in the middle of stuff that seems so complicated that it will take you forever to understand. If all else fails, attack the easy stuff on the outside. Do this even if the easy stuff seems to have nothing at all to do with your goal. The parts of the code that outputs characters or inputs data are usually easy to read. Continue carving away on anything that looks like it might shake loose. What this indirect attack does is *reduce* the size of what is left to a point where you can hack at it directly.

A big plus for the indirect attack is that it can show you the program author's style and where his head is at. Does he use self-modifying code? How does he handle multiple choice addresses? Does he use the indirect commands effectively and gracefully? Is he using mostly branches, or mostly jumps? How elegantly or how clumsily does he handle 16-bit addresses and long files? Does he extensively use the existing monitor, DOS, and BASIC subs, or is he reinventing the wheel? How clean is his organization? Is the program designed from the ground up for an Apple, or was it obviously modified from a program originally designed to run on some inferior machine? Answers to these questions can simplify very much the cracking of the rest of the code, since most decent programmers tend to be consistent in how they do things.

If the code seems ridiculously obscure, attack the fundamental subroutines. These subroutines are the ones that won the popularity poll (the ones with all the dots). The subs to hit first are those that will not call any other subroutines, but will go ahead and do direct and obvious things. Common things that fundamental subroutines will do include searching a long file for a value, calculating an address, or making a hex-to-decimal conversion.

Once you understand these fundamental subroutines, you don't have to go through them each time they crop up, since you know what they do. Create meaningful names for these fundamental subroutines and they will help you a lot in your attack. Then, "ripple" this new information back through the listing.

Asking for help is an obvious thing to do. There is nothing more infuriating than having an 8-year-old boy, just in from off the street, make some casual comment that completely sums up what it just took you months to find out the hard way. So discuss the target program and its attack. Don't only do it with "experts," but rap about it to anyone who will listen. Chances are their heads are in other places and might put things in a new light for you.

The *python force feeder* takes some special hardware, but it can be very effective. A force feeder is some hardware and software modifications that include a super-powerful bus driver, say a 74S245, or maybe three of them in parallel. When you tell it to do so, it substitutes its *own* code for what the computer is supposed to be working with.

For instance, even the old monitor ROM can't help plowing part of the display page, the first few keybuffer locations, and part of page Zero when it is activated. A sneaky programmer can hide things in plowable locations. But not so with a force feeder. Besides being able to force a monitor reset any time you like, a force feeder can substitute anything at any place in the program. It can also move copies of plowable locations to unplowable ones for analysis.

As a much simpler example of force feeding, consider the "top display line" copy protection hoax. What you do is switch to HIRES and, then, put a key jump or some other "magic" code that you want "hidden" on the top line of text display page One, starting at location $0400. This code is called early in the program and the program bombs if the code is not there. Naturally, the code gets erased immediately after use.

This, in theory, makes any messing with the program impossible. Any tampering at all will scroll up the display page and destroy the magic code. Sounds both bulletproof and infuriating.

In reality, this is only a "seven-second" copy protection. What you do is force feed the Apple by making it display only text page One, and this "hidden" code actually leaps out at you, shouting to be heard. To force feed the page One display on older Apples, remove integrated circuit F14 and ground pin No. 6 of the socket at F14. The hidden bytes will appear in Apple video-screen code, rather than op code, but if you got this far, that just adds to the fun.

Similar force-feeding games can be played with most of the Apple soft switches that are needed for analysis or debug.

Another handy debug trick is the *partial boot.* Instead of letting the target program completely boot, you only let it go so far and, then, analyze what you have. This catches code modules *before* they are moved to cover DOS, and so on. For instance, the program, *Pool 1.5,* is generally considered to have exceptionally good, or "three-hour," copy protection. But, use a partial boot and the "three-hour" protection drops down to a much more convenient "eighteen-minute" protection. More elegant "boot tracing can also be done.

The trick here is to carefully watch the disk drive with the cover off and time the different parts of the loading and protecting process.

By the way, there's one sure-fire way to read *any* disk at any time. Just glomp a *logic analyzer,* with a 6502 personality module in it, onto the CPU and you are home free. Unfortunately, you can buy a dozen Apples for the price of one better grade logic analyzer, so this ultimate weapon does not see much use.

Change detection is another interesting attack method. However, I haven't fully explored this one. What you do is dump part of memory, run a portion of

the target program, and then see what changed. By finding out how, when, and why that change took place, you can often gain all sorts of insight into what is going on.

Some day, I would like to build the ultimate change detector. This would take a DMA modification to the Apple that would let a *second* Apple or some type of dedicated hardware give you an instant and *separate* picture of memory activity while the main program was running. One display would show what the program was doing, while the second would show you each and every memory location of interest. Ideally, such a program should present any location or any block of locations that you want and would clearly identify them. With this ultimate change detector, you could actually *watch* the program while it was doing its thing.

A variable-speed feature would also be nice here, so you could slow down or stop key activities without waiting forever for them to get through a delay loop or whatever.

We've already seen how altering files can tell you lots of things in a hurry about your program. Sometimes you are shooting in the dark since some file locations may only rarely be used or might be used only in an obscure way. File changing is certainly worth a try.

If you are going to change the target program or interact with it, it might pay to put the program on your own assembler and create your own source code. This lets you add your own hooks and make changes of your own choosing inside the target program. The EDASM on the *DOS Toolkit* is ideal for this. Assembling your own source code backwards from the object program is quite a hassle, though, and you shouldn't try it unless you have pretty much cracked everything else. Disassembler programs, such as RAK-WARE's DISASM, are also available that will "capture" code for your favorite assembler.

Sitting on your program is often overlooked. Just walk away from the attack for hours or days, and things that should have been obvious all along will leap out at you. Let your subconscious work on the puzzles that are holding you up.

It works.

Another thing that can help is to try attacking a similar program, either by the same author or by one that does the same thing in a simpler or easier-to-understand way. The insights you get from one program will help you attack the other program.

Deciphering special codes may be needed in longer adventures. These codes are more often used to make code more compact than they are to purposely "hide" the meanings of what they hold. The trouble is that most compaction schemes used also do a most thorough job of masking everything that the file holds.

For instance, in *Zork,* the ASCII strings are compacted so that two bytes hold three characters. Some newer adventures use paired letter or similar codes to remove the redundancy from text messages so that long text files will fit inside the machine. This is how the *Collossial Cave* adventure from Adventure International manages to get everything that once demanded a mainframe dino into a 48K Apple without needing repeated disk access.

About the only way to attack these codes is to go into the code modules that decipher them. Then, decipher the decipheree. Single step, trace, or breakpoint access code modules till they show you how to read the file. Usually, there will be some obscure command or program feature that will do things a lot faster or simpler than the others. Trace this command or feature out and let it crack the code for you. The last resort, of course, is to give up. Go back and attack something that is simpler.

My first machine-language attack of a major program was Adam's *Pyramid*

of Doom. This was done on a wilderness firetower using nothing but a 6502 pocket card. It literally took all summer, but it led to this attack method, and there is no better way to learn machine-language programming.

CONVERGE ON YOUR GOAL

Just as soon as you have the structure pretty well defined and as soon as you have cracked most of the code modules, return to your original goal and solve that particular problem.

Our goal in HRCG was to find the scroll hooks. By now, they should leap out at you.

Just as the cursor is about to go off screen at $9208, a check is made to see whether scrolling or wraparound is to be used. If scrolling is active, $9213 does a jump to the scroll subroutine starting at $9221. Specifically, $9214 will hold the *low* address and $9215 will hold the *high* address of the scrolling subroutine.

Just change these hooks enough so that you can use your own scrolling subroutine.

Summing up . . .

The HRCG scroll hook is at $9214.
$9215 holds the address low of the scroll subroutine.
$9216 holds the address high of the scroll subroutine.
The existing scroll subroutine starts at $9221 and ends with $926F.

Easy, wasn't it?

If not, go through a few practice target programs and see how fast and powerful this method can be.

WRITE IT DOWN!

Surely you don't want to go through all this a second time on the same target program. So, carefully write down everything you learned in some form that works for you.

Make a clean copy of your analysis on the second listing you made. Also, make a neat new table of variables, a new cross-reference, and write a complete new script. Put most of this information onto disk so that you can have printable and updateable copies for later use. Use your word processor.

The insight that you have now will be long forgotten in a month. Be sure that you will be able to later recover what you already have done, and will be able to do so both quickly and hassle free.

Resist the urge to pull a "EUREKA! I have found it!" and run off with only your limited goal met. Do so, and the key information will disappear down the tube somewhere and all will be lost.

The following outline sums up all the steps involved in tearing into machine-language code. Go back over them, and you'll find three parts to the attack. First you *prepare* yourself, then you *attack* the target program, letting it reveal itself through its form and structure. Finally, you *follow up* the attack to reach your goal.

Here is a quick summary of the tearing method. . . .

TEARING INTO MACHINE-LANGUAGE CODE

PREPARATION

( ) Assemble the toolkit.
( ) Grok the program.
( ) Go to the horse's whatever.
( ) Set a limited goal.
( ) Empty the machine.
( ) Find where the program sits.
( ) List and hex dump the program.

ATTACK

( ) Separate action from bulk files.
( ) Paint subroutine returns green.
( ) Paint subroutine calls orange.
( ) Paint absolute jumps pink.
( ) Paint relative branches blue.
( ) Separate modules and stashes.
( ) Identify files and stashes.
( ) Attack variables and constants.
( ) Paint housekeeping yellow.

FOLLOW UP

( ) Make a list of variables.
( ) Write a script.
( ) Customize the attack.
( ) Converge on your goal.
( ) WRITE IT DOWN!

Practice makes perfect. Try it.

An obvious second program for your tearing attack would be FID on the DOS System master diskette. Try this one on your own and see how far you get. As a specific goal, find out how to use the code that tells you how much space you have left on a diskette

WILL THE REAL LISTING PLEASE JUMP OUT?

There are times when the disassembler in the Apple monitor lies like a rug.

A disassembler always assumes it is working with valid op codes. It starts with the first code byte it finds and, then, decides what operation the Apple is to do. Depending on the particular op code, one, two, or three bytes will be needed to complete the operation.

For instance, the CLC or clear carry command is an *implied* addressing instruction handled with a single byte. No further information is needed. The LDX #05 *immediate* command takes two bytes, one to tell you what to do and one to answer "How much?" The STA $4050 command uses *absolute* addressing and takes three bytes, one to tell us what to do and two bytes to answer "Where?" by giving us address low and, then, address high values.

Thus, a disassembler will automatically jump one, two, or three bytes to get to the start of the new instruction. **The disassembler always assumes it is working with valid code from a legal starting point.**

If either the starting point is wrong or if what is being disassembled is not legal code, the "lister" starts lying.

Suppose we have these bytes stashed in memory . . .

```
$0800- 80 8D AD 02 A5 18 EA
```

Here is what you get if you try to disassemble this code from various starting points. . . .

```
$0800- 80          ???
$0801- 8D AD 02    STA $02AD
$0804- A5 18       STA $18
$0806- EA          NOP

$0801- 8D AD 02    STA $02AD
$0804- A5 18       STA $18
$0806- EA          NOP

$0802- AD 02 A5    LDA $A502
$0805- 18          CLC
$0806- EA          NOP

$0803- 02          ???
$0804- A5 18       STA $18
$0806- EA          NOP
```

We see that we get a different disassembly every time, depending on where we start from. Which one is correct?

The correct disassembly is the one that begins with the first valid op code on the list. The first valid op code is often pointed to elsewhere in the program by a jump, a branch, a subroutine call, or an external entry point.

You can expect the "lister" to lie about one-half of the time when it comes out of a file or dead code and starts into legal code.

Usually the "lister" will correct itself after two or three wrong entries. So, you usually only have to worry about the first few entries into valid code.

If what you have just listed seems dumb, try listing from one above or one below where you think the legal code starts. Most of the time, there will only be one rational and sensible starting place and the valid code will leap out at you.

But remember that the "lister" will only tell the truth when it has both true code and a true starting point to work with.

SEEDS AND STEMS

To extend the life of a game-paddle connector that gets used a lot, plug the paddles into a 16-pin, premium *machined-contact* DIP socket. Then plug this socket into J14 on the Apple mainframe.

It pays to put sockets on all of your joysticks, paddles, and whatever, as well. Should a pin bend or break, repairs are far easier.

SEEDS AND STEMS

To edit a comment line in EDASM without having big holes chopped in it, use the "T" command to eliminate all tabbing.

To restore EDASM back to normal, either reboot, or else, use a "T14, 19, 29" command.

Tabs *must* be restored before assembly.