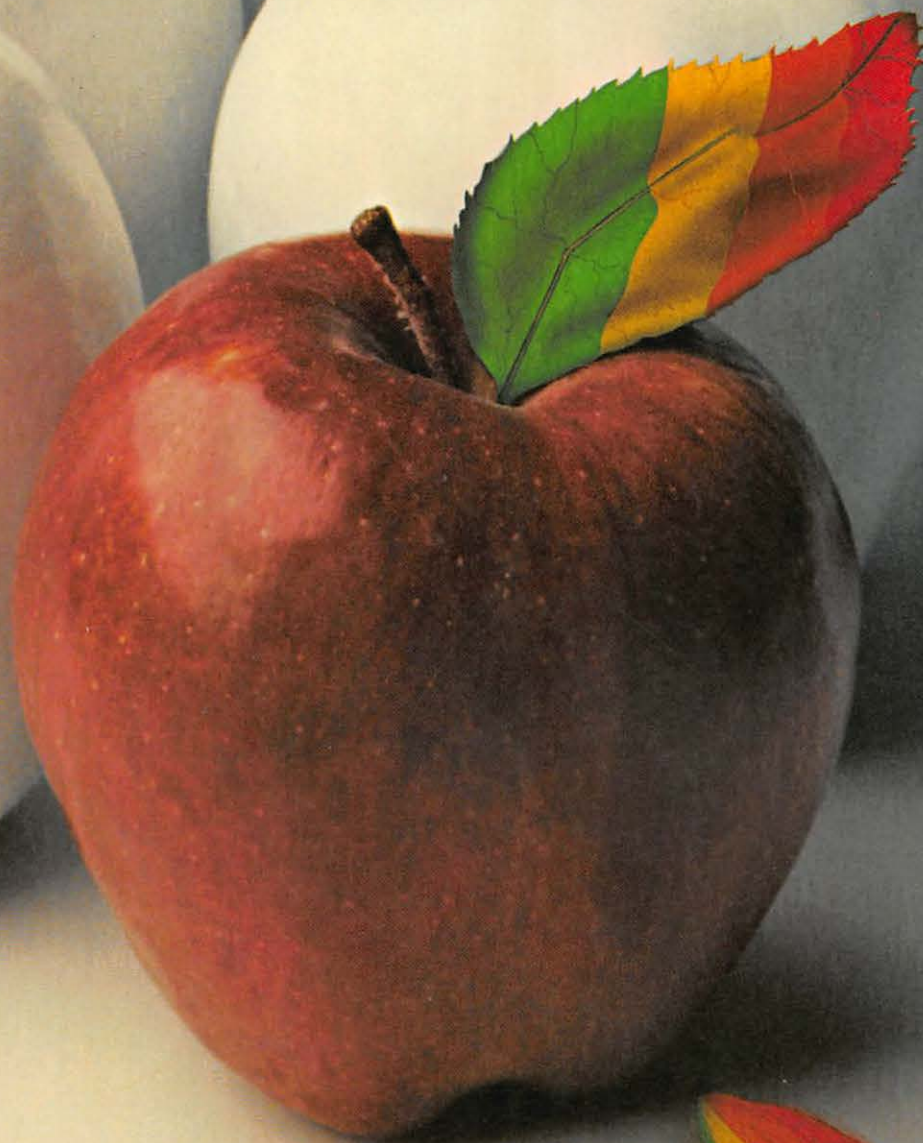


ENHANCING YOUR APPLE II®

VOL. 1

BY DON LANCASTER



ENHANCEMENTS INSIDE

- Two Glompers
- Software Color Killer
- Tearing Into Machine Language
- Field Sync
- Fun With Mixed Fields
- 121 Lo-res Colors
- The Glitch Stomper
- Gentle Scroll
- Fast Backgrounder

SAMS - NMM46
\$15.95

ENHANCING YOUR APPLE II®

VOLUME 1

DON LANCASTER heads *Synergetics*, a new-age design and consulting firm involved in microcomputer applications and electronic design. He is well known as the author of the classic *CMOS* and *TTL Cookbooks*. His many other books and hundreds of articles on personal computing and electronic applications have set new standards as understandable, useful, and exciting technical writing. Don's other interests include ecological studies, firefighting, cave exploration, bicycling, and tinaja questing.

Other SAMS books by Don include *Active Filter Cookbook*, *CMOS Cookbook*, *TTL Cookbook*, *Cheap Video Cookbook*, *Son of Cheap Video*, *TV Typewriter Cookbook*, *The Hexadecimal Chronicles*, *Don Lancaster's Micro Cookbook*, and *The Incredible Secret Money Machine*.

ENHANCING YOUR APPLE II®

VOLUME 1

by Don Lancaster

*Apple and Apple II are registered trademarks of Apple Computer Inc., Cupertino, CA.

Copyright © 1982 by Howard W. Sams & Co., Inc.
Indianapolis, IN 46268

FIRST EDITION
SECOND PRINTING—1983

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. While every precaution has been taken in the preparation of this book, the publisher assumes no responsibility for errors or omissions. Neither is any liability assumed for damages resulting from the use of the information contained herein.

International Standard Book Number: 0-672-21846-1
Library of Congress Catalog Card Number: 82-50013

Edited by: *Frank N. Speights*
Illustrated by: *T. R. Emrick*

Printed in the United States of America.

CONTENTS

INTRODUCTION	7
---------------------------	----------

ENHANCEMENT 1

TWO GLOMPERS	11
---------------------------	-----------

Two methods to eliminate hassles over rf changeover switches.

ENHANCEMENT 2

PROGRAMMABLE COLOR KILLER	17
--	-----------

Simple modification that eliminates color fringes on HIRES text displays under software control.

ENHANCEMENT 3

TEARING INTO MACHINE-LANGUAGE CODE	29
---	-----------

An astonishingly fast and simple way to tear apart someone else's machine-language program.

ENHANCEMENT 4

FIELD SYNC	89
-------------------------	-----------

One wire add-on for stunning new animation, game, video, and control possibilities.

ENHANCEMENT 5

FUN WITH MIXED FIELDS117

Mix or match text, HIRES, and LORES any place on the screen, fast and flicker-free. Also reveals the secret of the 121 LORES colors. Or does it?

ENHANCEMENT 6

GLITCH STOMPER159

Add-on adaptor that further improves field sync and mixed field switching.

ENHANCEMENT 7

GENTLE SCROLL171

Easy to use program gives you gentle or crawling scrolls that are easily read while in motion.

ENHANCEMENT 8

FAST BACKGROUNDER201

Utility program that gives you hundreds of HIRES colors or zillions of background patterns.

INDEX.....229

INTRODUCTION

If you use your Apple II personal computer often enough, long enough, and late enough, eventually your Apple may decide to reveal "IT" to you.

"IT" is a series of revelations on where we came from, where we are, and where we are heading. Most often, your Apple will reveal "IT" to you very early on some morning in your second or third year of Apple use.

The "IT" revelation is for real. Just ask anyone who has worked with an Apple long enough. The wry smile and knowing nod tells all.

If my Apple will forgive me, I'd like to reveal some of the present "IT" messages to you, for they say a lot . . .

- * * * -

The Apple II is far and away the single most powerful tool **ever** put in the hands of many individuals on an uncontrolled and unregulated basis. The new personal freedoms and the potential opportunities that result from this are almost beyond belief. It's now a whole new ball game, a jump into hyperspace from where we are. The Apple II is far more significant and it will have a vastly greater impact than such short term frivolities as the automobile and television—and, possibly, even more than the printed word itself. Even time may eventually be measured as "BA" and "AA," split at that magic date in the spring of 1977.

- * * * -

The Apple experience is absolutely and totally unique. There is no other product available anywhere, at any price, that adapts itself as easily, as well, and as conveniently to whole new worlds of strongly user-oriented applications. The unbridled potential of Apple's sixteen I/O sockets is utterly awesome.

- * * * -

Future historians will recognize the Apple II as the DC-3 of the microcomputer revolution.

- * * * -

There are several reasons for Apple's success. The first is simply being in the right place at the right time. The second is blind luck. But, the third and most important reason is that Apple has lacked "us-versus-them."

Apple has always treated its users like they were friends and not the enemy. They didn't seal the works up, or keep you out of the system monitor, or hide their documentation under an armadillo somewhere, or try to prevent you from doing easy add-ons, enhancements, or expansions. Apple didn't force software people to buy an expensive development system. Instead, the Apple II itself is its own superb development tool. There's no fancy modules or other restraints that keep anyone from getting into the Apple support business. In fact, the widest possible number of Apple hardware and software suppliers was encouraged from the start. And the "our engineering department not only knows best, but they are God" attitude of other manufacturers is simply not there with Apple.

- * * *

The garbage-to-good ratio of Apple software is skyrocketing. The quantity of truly astoundingly atrocious Apple software is now running at least a thousand times ahead of the useful and reasonable stuff. And, it's getting worse.

Garbage software fails to use the unique Apple resources to the utmost. This type of software may be as snotty as most of the dino stuff. It may use fancy packaging and/or expensive promotion to try and make up for sloppy and poorly thought-out coding. Garbage software is usually locked, so that the user cannot back up or modify the program to suit his own needs. It is usually slow and awkward to boot. Garbage software also usually demands oddball codings and disk formats that guarantee incompatibility with everything else.

Garbage software is caused by people who are greedy, sloppy, dumb, inexperienced, in a hurry, or all five. Garbage software is not user oriented. It is overpriced. It has miserable error recovery. It tries to solve some small specific problem, rather than being a generalistic tool that can handle a broad class of problems. Garbage software steals the ideas of others, but can only come up with a second-rate result at best. It attacks problems that not only do not need solving, but which shouldn't be solved at all with a computer.

Please, if you are going to write this sort of trash, go get yourself a different brand of personal computer. We don't need you and we definitely don't want you.

- * * *

Any attempt whatsoever at copy protection will hack off and inconvenience your legitimate users and it will dramatically increase the number of bootleg copies of your program in circulation. It will also price your program out of the market.

A user of software **demands** the absolute right to make backup copies of everything he buys, and **must** have the right to examine and modify all coding in that software so he can meet his own needs. These are mandates.

The big thing about copy protection is that it doesn't. A year's effort by a crackerjack military cryptography team can usually be undone in fifteen minutes, between klingon zappings, by your average fourteen-year-old. And, morality and economics aside, one fact stands out

Undoing copy protection is fun!

Not only is it fun, but cracking the uncopyable is about the most challenging and most rewarding thing that you can possibly do with your Apple. And, the things you learn along the way are exactly the skills that you will need to become a really great programmer. So, I guess we should all be thankful for the copy-protection people since they are giving us all this fascinating entertainment and superb training at an unbeatable price.

- * * *

Today's best and brightest Apple programmers are stumbling around in the dark. Proof of this is that a program with a *Peelings* rating of "AA" today barely earns a "B" next year, and drops to a "D" or unfit-for-use "F" the next.

Surely, by now, the message is in: it is absolutely impossible to write a great program in BASIC. Pascal, of course, is so bad that it is beyond the pale. Great programs **must** be written either wholly or in part in machine language so they can use Apple's resources to the utmost, at the fastest possible speed. Check *Softtalk's* top thirty. At this writing, thirty out of thirty are either written totally in machine language or make extensive use of machine-language sequences.

It's scary to think of what a really good programmer, who truly understands his Apple, will be able to do with this machine. It hasn't happened yet, but watch out. It is only a matter of time.

- * * *

We are only now beginning to find out about some secrets of the Apple that everybody should have known about way back in 1977. For instance, we know now that we can have many different fonts of upper- and lower-case characters without using special hardware. We know that we can easily do an 80-character line with zero additional hardware, and that lines above a hundred characters are possible. We know that it is easy to quadruple the HIRES virtual resolution of the Apple.

We know, of course, that there are hundreds of colors available either in HIRES (high resolution) or LORES (low resolution). We also know that we can mix and match HIRES, LORES, and text anyway that we like—anywhere on the screen. We know that we can do an exact, jitter-free, software lock to video timing for video wipes and precision light pens. Once again, these require no special hardware. And, of course, we know we can gently and legibly scroll the characters up the screen. We know that the early graphics mappings were much slower than necessary. And, we know our Apple can directly interface robotics and appliance controls—again, without special hardware.

The big question is: "How much **don't** we know that we should have known back in 1977?"

* * * * *

This *Enhancing Your Apple II*® series is intending to try and bring everybody up to 1977. We want to try and understand what we really have in the Apple II, and what its **real** capabilities and limits are.

Each enhancement is designed to show you something about some small corner of your Apple. While a typical enhancement will combine some simple new hardware with a machine-language driver or two, just about anything at all is likely to crop up. We have tried to mix simple and advanced enhancements together, so there will be something here for you, regardless of how much Apple experience and expertise you now have.

We also have "unbundled" everything for your convenience and cost savings. Each enhancement is in four pieces. The first part of an enhancement is the complete story and listings here in this volume. Secondly, there is a companion diskette you can order using one of the cards in the back of the book. This diskette includes copies of all the code used in this volume and more. Machine-language codings include full source documentation under both EDASM and the S-assembler while BASIC Programs include full documentation. Naturally, you are free to copy, adapt, and modify this standard DOS 3.3 diskette to your heart's content, so long as you do so only for your own use.

Thirdly, there is a parts kit you can order that includes everything you will need to make all of the hardware modifications involving all the enhancements in the entire book. Lastly, and most important, there is also a feedback card. This card is needed for your participation in some of the enhancements. It also registers your name for updates and corrections, and it "closes the loop" so we can offer the best possible Apple enhancements in future volumes.

Should you be interested, we will add an on-line update and bulletin board service to this feedback process. Let us know if you have a modem and want to participate.

Taking a quick look at this volume, we start out with a pair of simple *glombers* that anyone can build. These solve completely the video changeover-switch hassles that you might have when using an rf modulator. Enhancement 2 is a software *color killer*. A plug-in reversible hardware mod lets you eliminate all color fringes from black and white high resolution (HIRES) displays under software control. You can now combine color and black and white displays in the same program without any set adjustments.

The real heavy of this volume is Enhancement 3. Here we find a method for tearing apart someone else's machine-language program that is astonishingly fast and super easy. It may take you years to fully explore and fully comprehend the implications of this single enhancement.

Another heavy appears in Enhancement 4. We add a lone wire here to give you a way to exactly lock your programs to video timing. And, we mean exact. The locking is done jitter free, which opens up bunches of new applications, such as video wipes and precision hardware-free light pens.

Want to mix and match HIRES, LORES, and text anywhere on your Apple screen? This is trivially easy once you master Enhancement 5. We also see how to tap the hundreds of available LORES colors in this enhancement. A companion hardware mod called a *glitch stomper* appears as Enhancement 6. This one makes displays that switch modes on screen operate even better.

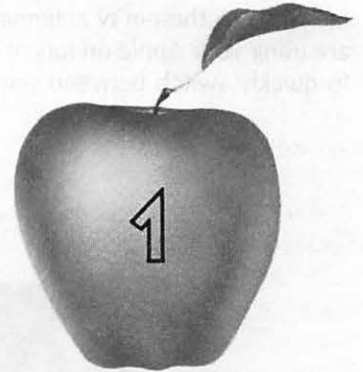
How about a *gentle scroll*, where the characters move smoothly up the screen rather than jumping up illegibly like they do now? This one's done in Enhancement 7. It's super smooth, and completely glitch free. The final enhancement in this volume shows us how to pick up hundreds of different HIRES background colors, and how to put them down seven times faster than you might think possible.

That's about it for Volume 1. But, we have some really great stuff on tap for future volumes. A sneak preview appears at the end of this volume. This series is open-ended. As long as there are new things to learn about your Apple® and new ways to do them, we will try and show you how to do them.

Oh, yes. Some legal beagle somewhere will probably get bent out of shape if I don't tell you that Apple is a registered trademark of some outfit in California whose name I don't recall just now, and that everything here is pretty much my own doing, and is done without Apple's knowledge or consent.

DON LANCASTER
Summer 1982

Enhancement



TWO GLOMPERS

Here's two different ways to eliminate the hassles caused by the changeover switches on rf modulators. One glomper is portable. The other is more or less permanent.

TWO GLOMPERS

Have you ever been infuriated by that #\$\$%# & changeover switch on your Apple's rf modulator?

If you can find a flat place to stick it on the back of your tv, it's stuck there for good. If you can't, the switch quickly does a *Kamakazi* act, strangling itself on its own leads. And, this switch is very difficult and inconvenient to move from tv to tv.

This changeover switch is totally useless. It doesn't even do what it is intended to do, since you can easily leave a built-in antenna in place and radiate your video all over everywhere.

Here are two much saner antenna *glompers* that you can use instead. Either glomper version will quickly and easily fit any tv. They are cheap and simple enough that you can build lots of them.

Most rf modulators provide a phono plug input to the changeover switch. This phono plug fits a standard RCA phono jack. Both glompers start with an RCA phono jack and adapt it so that it is super easy to connect to your tv.

Our glomper of the first kind is shown in Fig 1-1. This one mounts a phono jack on a clothespin tv antenna connector. This style glomper is best when you are using your Apple on lots of different tv sets at different times, or if you want to quickly switch between sets.



Glomper of the first kind is made from a phono jack and a clothespin connector. Use it if you often change or test different tv's.

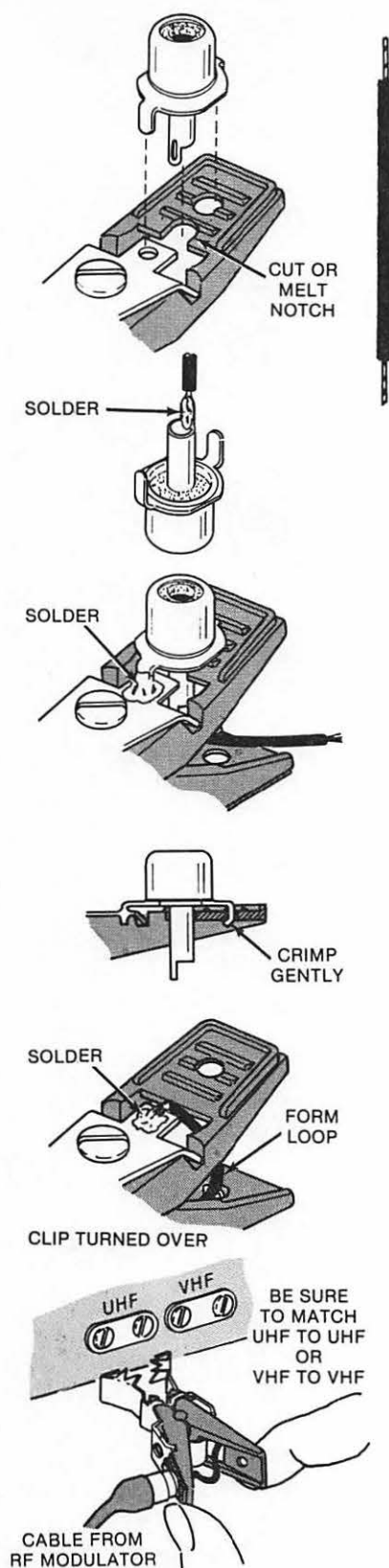
Fig. 1-1. A glomper of the first kind.

The glomper of the first kind is also very useful if you are buying a color tv and want to take your Apple to the store in order to compare lots of models to see which handles Apple video the best.

Here are the parts you will need for a glomper of the first kind

PARTS LIST FOR A GLOMPER OF THE FIRST KIND
() Clothespin-style tv antenna connector.
() RCA phono jack, vertical pc-mount style.
() No. 22 stranded hookup wire, insulated, 2 inches long.
() Solder flux (optional).
() Short piece of electronic solder.

And here is how to build one



INSTRUCTIONS FOR BUILDING A GLOMPER OF THE FIRST KIND

1. Try to fit a RCA upright pc phono jack to the two holes in the clothespin antenna connector as shown. One hole is in the metal and one is in the plastic.

If the center connection of the phono jack interferes with the plastic handle, remove some of the plastic as shown. Use a knife, a file, or simply melt the plastic with a soldering iron as needed.

2. Strip 1/4 inch of insulation from both ends of a 2-inch-long piece of No. 22 stranded wire.

Solder one end of this wire to the center conductor of the phono jack. Make sure the phono plug fits after soldering.

3. Carefully and thoroughly clean both the phono jack and the clothespin connector at the point where they are to be soldered together. Add a drop of super-safe electronic solder flux if it is available.

Solder the phono jack to the antenna clothespin connector as shown. Use a soldering gun or a medium (100 watt) soldering iron.

4. Gently crimp the unsoldered ear of the phono jack against the plastic of the clothespin connector.

Be sure the jack has cooled before you do this.

5. Carefully clean the other side of the clothespin connector. Add a drop of super-safe solder flux if it is available.

Solder the free end of the stranded wire to this side of the clothespin connector.

6. Flex the clothespin a few times to be sure it works smoothly. Remove any remaining solder flux.

This completes your assembly.

7. To use your glomper, plug the pin plug on the rf modulator cable into the jack on the glomper, and clip the glomper on the proper antenna terminals.

ALWAYS REMOVE ALL OTHER OUTSIDE AND INTERNAL ANTENNA CONNECTIONS WHEN USING THIS GLOMPER!

Fig. 1-2. How to build a glomper of the first kind.

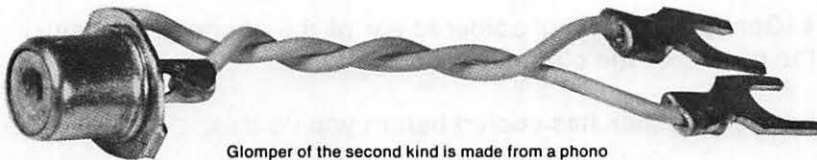
Using these tools . . .

TOOLS NEEDED TO BUILD EITHER STYLE GLOMPER

- () Needle nose pliers
- () Diagonal-cutting pliers
- () Wire stripper
- () Soldering gun or medium soldering iron
- () Ink eraser or steel wool
- () Terminal crimper (optional)
- () Small vise or clamp

Construction details are shown in Fig. 1-2. You mount the phono jack on the plastic handle of the clothespin connector. Be sure to use a stranded wire between the center of the phono jack and the other arm of the clothespin. Solid hookup wire will soon break if you flex it too often.

You will also want to carefully clean both the phono jack and the clothespin connector before soldering. This can be done with an ink eraser or steel wool. If you have some, a drop of "super-safe" electronic solder flux will make things much easier. *Do NOT use any other type of flux!*



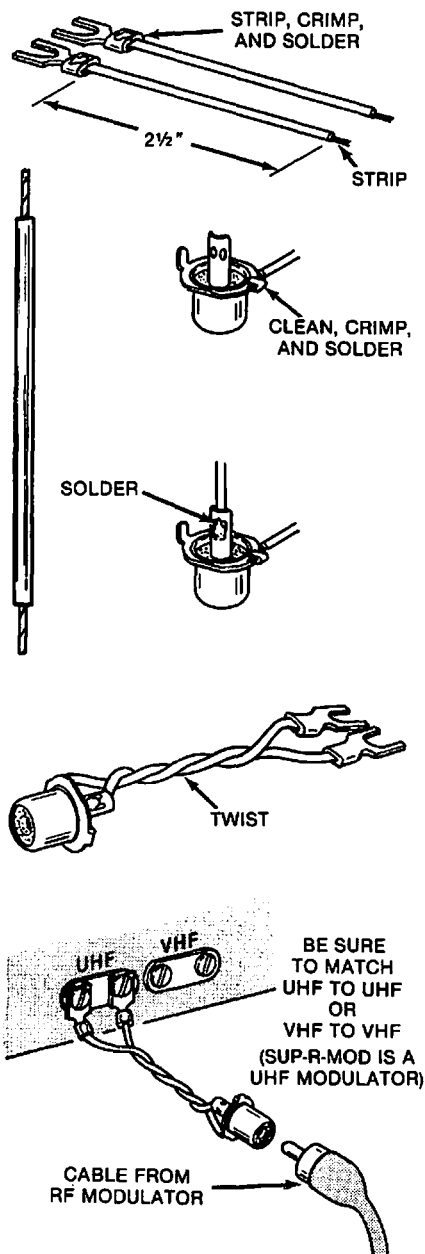
Glomper of the second kind is made from a phono jack and spade lugs. Use this glomper for more permanent connections.

Fig. 1-3. A glomper of the second kind.

Our glomper of the second kind is shown in Fig. 1-3. This one is designed to be permanently connected to a tv. It is nothing but an RCA phono jack and two stranded wires ending in spade lugs. Here's the parts you will need . . .

PARTS LIST FOR A GLOMPER OF THE SECOND KIND

- | | |
|--|--|
| () RCA phono jack, vertical pc-mount style. | () No. 22 stranded hookup wire, insulated, 5 inches long. |
| () Crimp-on spade lugs (2 needed). | () Short piece of electronic solder. |
| () Solder flux (optional). | |



INSTRUCTIONS FOR BUILDING A GLOMPER OF THE SECOND KIND

1. Cut two pieces of No. 22 stranded hookup wire 2½ inches long and strip ¼ inch of insulation from each end.

Crimp and then solder one spade lug to one end of each wire.

2. Carefully polish the inside of one leg of an upright pc phono jack. Then roll this leg over onto the free end of one of the wires.

Add a drop of super-safe solder flux if available and solder wire to jack as shown.

3. Solder the free end of the remaining wire to the center conductor of the phono jack.

4. Carefully inspect the wiring to be sure there is no short between the outside and the inside of the phono jack.

You might like to bend the remaining leg of the phono jack inward for better appearance.

5. Twist the two leads together four or five times and arrange the lugs as shown.

This completes your assembly.

6. To use your glomper, connect the spade lugs to the tv's antenna terminals. Plug the pin plug on the rf modulator cable into the jack on the glomper.

ALWAYS REMOVE ALL OTHER OUTSIDE AND INTERNAL ANTENNA CONNECTIONS WHEN USING THIS GLOMPER!

Fig. 1-4. How to build a glomper of the second kind.

Complete construction details are shown in Fig. 1-4. Again, be sure to use stranded wire, carefully clean before soldering, and use a drop of "super-safe" flux, if you have it available.

If you have several of your own tv sets in use, put a glomper of the second kind on each one.

Be sure to connect the output of a uhf rf modulator (such as the SUP-R-MOD) to the uhf antenna terminals, or the output of a vhf modulator (some others) to the vhf antenna terminals.

Note also that there are two RCA jacks on the SUP-R-MOD. The one you want goes down into the rectangular shielded box. The other one is not normally used and does *not* output any rf signal. It pays to label these two jacks "RF OUT" and "VIDEO THRU" by writing on the inside wall of your Apple with a *Sharpie* or similar heavy pen.

Regardless of which style glomper you use, please obey this rule

**DO NOT EVER CONNECT THE RF
OUTPUT OF AN APPLE SYSTEM
TO A BUILT-IN ANTENNA, AN
OUTDOOR ANTENNA, OR A
CABLE TV LINE.**

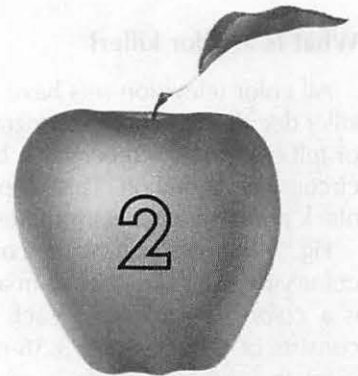
If you are using both an outside antenna and a glomper, you should put the outside antenna line on a clothespin connector as well. If you have cable service, always unscrew the cable connector before using your Apple.

A glomper of either style *must* be used with an rf modulator. Just because the Apple's baseband video output jack is also a phono jack, don't expect any useful results if you try to feed raw video into the antenna terminals of a tv set. While you might actually get something on the low channels if you try this, in no way will the display be stable or useful.

There is one other minor use for your glompers. Many of the music synthesizer cards output via a phono jack but can directly drive a speaker. If your speaker enclosure has a pair of screw terminals on the back, just use either style glomper to get from the music card to the speaker 🍏

**A complete set of all parts needed
to build two glompers of each
style is included in the companion
parts kit to this volume.**

Enhancement



PROGRAMMABLE COLOR KILLER

A simple, cheap, and reversible hardware modification that eliminates color fringes on text and removes unwanted color lines in HIRES displays. You now have a choice of black and white or color display under program control.

PROGRAMMABLE COLOR KILLER

Did you ever wish that you could eliminate those color fringes on Apple's HIRES text displays? Or, be able to get rid of the vertical color lines that sometimes mess up an otherwise stunning white-on-black HIRES display? Or, be able to do LORES special effects, where you switch from colors to grey patterns and back again?

Well, it's going to cost you. Around \$1.10 and 15 minutes of your time.

If both of these are within your budget, you can easily add your own software-controlled *color killer*. The mod will work on any but the oldest revision "0" Apples, and if you are careful, it will be completely reversible and won't void your warranty.

After the mod is completed, a single command inside your program can give you a choice of color or true black and white displays on your color tv or color

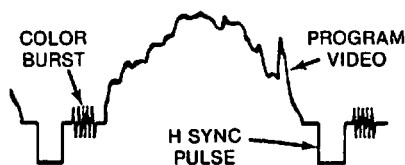
monitor. The software control commands are either a simple POKE or a BIT test and may be done in any language. The color killer shuts off on an autostart reset and stays disabled until you activate it, so it stays invisible until you want it.

Let's do it.

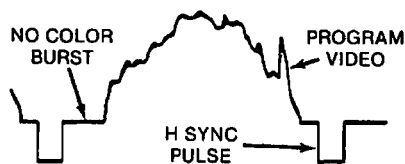
What is a color killer?

All color television sets have a built-in circuit called a *color killer*. The color killer decides whether the program material is being received in black and white or full color. If it is receiving a black and white signal, then the color decoding circuits are turned off. This keeps annoying color lines, snow, and fringes out of black and white program material.

Fig. 2-1 shows us how the color killer inside your color tv works. In a typical color video signal, either from a tv station or from your Apple computer, there is a *color burst* following each horizontal sync pulse. This color burst usually consists of 8 cycles of a 3.58-megahertz sine wave. When present, the color burst is used to provide a reference for use by the color decoding circuitry inside the television. This arrangement lets the circuitry tell one color from another.



In a typical color signal, a color burst is provided on the "back porch" of the horizontal sync pulse. This burst provides a color reference and deactivates the color killer, allowing a color display.



In black and white video, the color burst is absent. This activates the color killer in a color set and switches to a black and white only display.

Fig. 2-1. How the color killer, which is present in all color sets, can tell color from b/w video.

The color killer looks for these bursts, and if it doesn't find them, it defeats the color processing so that the set displays a black and white picture with no color fringes. If the burst is present, the color killer lets the color processing circuits do their thing, and you get a full color display.

Unfortunately for us, the color killer in your set is rather slow. It takes a fraction of a second for it to work. The reason for this is that in a fringe reception area where you get a weak or snowy picture, a color killer too fast or too sensitive would continuously switch from black and white to color and back again as the signal quality varied. This would really foul things up.

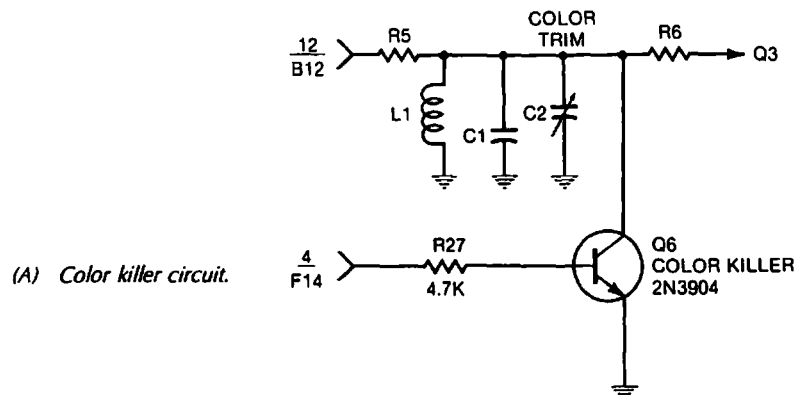
What this means is that, as Apple people, we can only expect a color killer to be on or off. We can't expect it to switch back and forth during a single field to give us, say, full-color mixed graphics with a true black and white four-line message on the bottom. This we cannot do. But we can easily switch from black and white to full color, and back again, anytime we are willing to have the entire screen be one or the other.

The modification

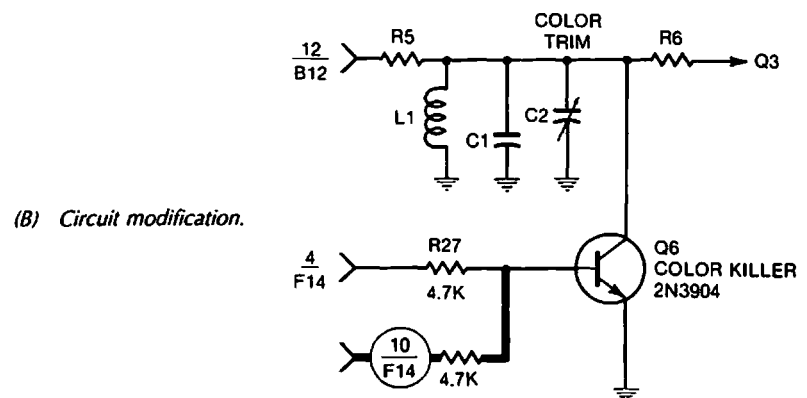
As Fig. 2-2 shows us, all but the earliest Apples have an automatic color killer circuit built into them. This circuit (Fig. 2-2A) consists of a 2N3904 transistor (Q6) and a 4.7K resistor (R27). Whenever you are in the full text mode, the point driving the 4.7K resistor goes positive, which turns Q6 ON, which shorts out the color-trimming circuit. The short, in turn, eliminates the color burst from the output. In anything but the full text mode, the point driving the 4.7K resistor goes low, which turns transistor Q6 OFF, allowing full color.

This existing hardware color killer on your Apple works well with most better-grade color tv's, but it only works on full text displays done the "old" way. It will not work on a text display on either HIRES page or on mixed graphics. Nor will it work on a HIRES graphics display that is supposed to be all black and white. Nor can it work on a LORES display.

Fig. 2-2B shows us how to add one resistor to give us software-controlled color killing. We add a second 4.7K resistor so that it also goes to the base of transistor Q6. This resistor comes from the source of AN1. Make AN1 high, and you kill the burst and get true black and white displays. Make AN1 low, and you get full color on everything but text displays.



Existing color killer circuit in your Apple works only in full text mode.



New resistor gives you software-controlled color.

Fig. 2-2. A single resistor is all you need to add a software color killer.

Rather than grab AN1 at the game paddle, we get it from its source, pin 10 of F14. This prevents accidental pull-outs when the game paddles are removed.

The autostart ROM always drives AN1 *low* on a cold start reset. This makes the color killer invisible until used. Note that there are errors in some early Apple reference manuals involving what happens during a cold autostart reset. I have found that what really happens is that AN0 and AN1 go *low* and AN2 and AN3 go *high* on a cold autostart reset. You should correct pages 36 and 143 of your Apple reference manual if they do not agree with this.

Logically, this is a NOR circuit because switching to the text screen OR turning annunciator AN1 ON activates the color killer and puts the colors OFF.

Building it

Here's the parts you'll need

PARTS LIST FOR SOFTWARE COLOR KILLER
() 16-contact quality DIP socket, machined-pin style.
() Pin from machined-pin DIP socket.
() 4.7K, 1/4-watt resistor (yellow-violet-red).
() Short piece of electronic solder.

And, here are the tools you will need

TOOLS NEEDED TO BUILD SOFTWARE COLOR KILLER
() Needle nose pliers
() Diagonal-cutting pliers
() Wire stripper
() Small soldering iron, 35 watt
() Any old 14- or 16-pin DIP integrated circuit
() Small vise or clamp

Fig. 2-3 shows us the area of the Apple's board that we are going to work with. The color killer transistor Q6 is located above integrated circuits F13 and F14.

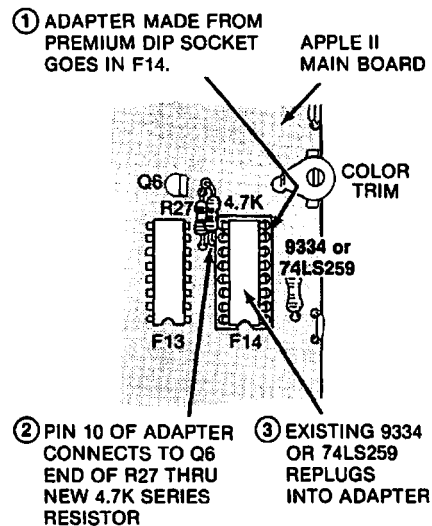


Fig. 2-3. Pictorial shows color killer hardware additions. Single pin socket at Q6 allows easy removal.

See the APPLE board parts locations sidebar located at the end of this enhancement for more details on how to locate a part on your Apple's main circuit board. There is a grid numbered from 1 to 14 across the bottom, and lettered A through K up the left-hand side. Letters "G" and "I" are omitted to save on confusion. Location "F14" is the place where the "F" row and the "14" column cross. Note that F14 is NOT the fourteenth component in that row. Some ICs are larger than others and other locations may not be used at all. Only the grid location counts. Note that the numbers across the top of the circuit board (not shown) refer to the I/O slots and have nothing to do with the grid. Note also that there are other resistors and transistors elsewhere on the circuit board. These have nothing to do with the color killer and must not be altered.

Your mod consists of adding an adapter socket to F14 that routes a 4.7K resistor from AN1 to a new single-contact jack added to the transistor end of R27. Fig. 2-4 gives the assembly details. Be absolutely certain you are using a premium 16-pin DIP socket with *machined-pin contacts*. These may safely be plugged into another DIP socket without damage. Also, be certain to keep all solder off of the actual part of the pins that must fit the socket at F14, and be careful not to melt the plastic part of the socket.

Should you ever want to undo the color killer, just reverse the above procedure. If you ever need some warranty repair, carefully untack the single pin socket as well, and clean up any remaining solder with a solder sucker or desoldering braid. Strictly speaking, the mod does void your warranty, but you can easily disguise the fact that it ever was made.

Always be careful when you insert or remove anything. See that the color killer mod stays in place. A dab of hot glue, silicon rubber, or other "semi-permanent" sticky stuff might be a good idea here to hold the killer socket to the pc board. Be careful, of course, to keep any gunk out of the socket pins.

Note that a cold start RESET using the autostart monitor will also turn the color killer off, but that an autostart RESET to some BASIC program usually will not. Thus, the color killer will normally come up as OFF if you don't mess with it, just like it wasn't there.

Here's how to install your software color killer . . .

INSTALLING THE SOFTWARE COLOR KILLER

1. Turn the power off and unplug both ends of the Apple power cord.
2. Remove the 74LS259 or 9334 integrated circuit at F14, using an IC puller if you have one available.
3. Plug the 74LS259 or 9334 into the color killer's 16-pin socket. Be sure the notches on both the socket and the IC point in the same direction.
4. Plug the color killer's 16-pin socket into the Apple mainframe at F14. Be sure that the resistor points away from you and that the notch on both socket and IC points towards the keyboard.
5. Plug the floating resistor lead into the single contact socket at R27, using needle-nose pliers.
6. Reconnect the power cord and apply power. Run the COLOR KILLER DEMO program for checkout.

INSTRUCTIONS FOR BUILDING COLOR KILLER

1. Take a single pin from a machined-contact socket and bend it as shown. If you destroy a socket to get one of these pins, melt the pins out rather than stressing them.

Plug the 4.7K resistor into the single pin socket as a temporary "handle." Then, tin only the bent part of the socket by applying a very small amount of solder.

Do NOT solder the resistor to the socket!

2. Find the end of resistor R27 nearest Q6 on the Apple main board and tin the lead of R27 here by applying a small amount of solder.

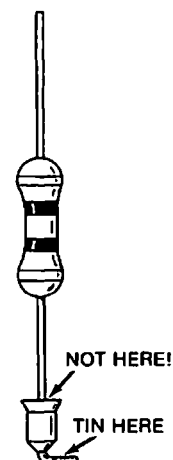
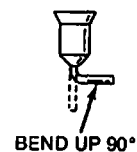
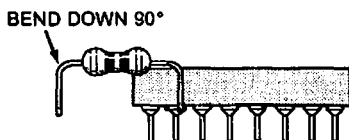
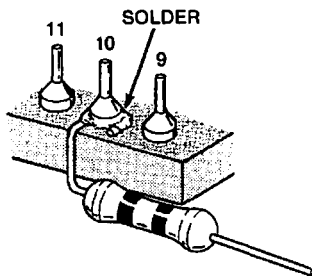
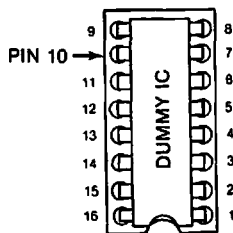
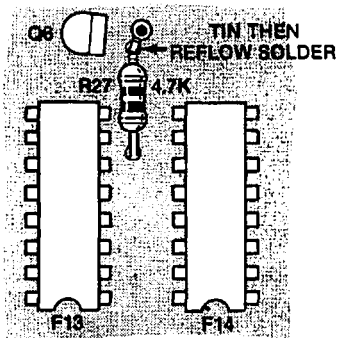


Fig. 2-4. How to build your color killer.



Then, reflow solder the "tail" of the single pin socket to the Q6 end of resistor R27. The tail should point towards the keyboard.

Remove the resistor "handle" after the solder cools.

3. Plug any old nonvaluable 16-pin DIP integrated circuit into the 16-pin machined-contact pin socket. This will keep the contacts aligned should the plastic soften.

Identify pin 10 by inking the plastic.

Note that this **MUST** be the type of premium socket that has small machined-pin contacts that are safe to plug into another socket.

Secure the socket in a small vise.

4. Cut both leads of the new 4.7K resistor to 7/16 inch length. Form a bend and a loop in one resistor lead as shown. Secure the resistor in this position against the socket somehow.

Solder the resistor loop to pin 10 of the socket, **EXACTLY** as shown.

Be sure that no solder gets onto the part of the pin that must plug into the Apple, and that there are no shorts to adjacent pins 9 and 11.

5. Remove the nonvaluable IC and set it aside and out of sight.

Position the socket as shown. Then bend the remaining resistor lead downward as shown.

This completes the assembly.

Refer to the text for installation and checkout.

Fig. 2-4 Cont. How to build your color killer.

Here's how you use your software color killer . . .

**USING THE
SOFTWARE COLOR KILLER**

From machine language —

BIT \$C05B turns killer ON

BIT \$C05A turns killer OFF

From either BASIC —

POKE — 16293,0 turns killer ON

POKE — 16294,0 turns killer OFF

There could possibly be a conflict between the color killer and anything else that is plugged into the game socket that uses AN1 as an output. Note that there would be no conflict on a text display or on a display routed to a black and white monitor. There may not be a conflict if the other use of AN1 is brief or rare, or if it takes place in the text mode. Naturally, if you must have AN1 for a conflicting use, you can always remove the socket and its resistor.

While a BIT test is the most "proper" way of switching AN1 from machine language, any old mode that addresses these locations, such as LDA \$C05B or STY \$C05A, may be used.

But . . .

**Do NOT disconnect the color killer
resistor without removing the DIP socket!
A floating resistor lead can cause damage
if it happens to hit anything.**

Program 2-1 is an Applesoft demo program that makes a pretty HIRES picture for you and alternately switches the color killer OFF and ON every few seconds. Watch particularly the effect on the slightly slanted lines in the middle of the image and the color fringing on the mixed graphics. Besides the Applesoft language, Program 2-1 needs the color killer hardware modification as described in this enhancement.

If you hand load this program, be sure to include the space following the "ON" in line 450. Note also that a color killer ON should give you black and white and a color killer OFF should give you a full color display.

You can use this demo to test your programmable color killer. The program works by first setting up a HIRES picture in the mixed graphics mode and adding some color blocks. Then, it turns the color killer OFF by poking AN1 to a zero, and writing the word "OFF" to the text area. It then delays for a few seconds, pokes AN1 to a one, and writes the word "ON." The process keeps repeating till you end the program with a CTRL-C or a RESET.

Some very old, cheap, or otherwise scungy color tv sets may ignore the color burst and do their color killing based on whether there is lots of energy in the "color band." Since Apple *always* has lots of energy in the "color band," these sets will ignore any color killing commands. On a set like this, the old hardware color killer won't work either. On these sets, you are stuck with the old method of manually backing all the color controls off.

PROGRAM 2-1 COLOR KILLER DEMO

LANGUAGE: APPLESOFT

NEEDS: COLOR KILLER HARDWARE
MOD OF ENHANCEMENT #2

```

10 REM *****
12 REM *
14 REM *   COLOR KILLER   *
16 REM *       DEMO       *
18 REM *
20 REM *   VERSION 1.0   *
22 REM *   (11-4-81)    *
24 REM *
26 REM *   COPYRIGHT 1981 *
28 REM * BY DON LANCASTER *
30 REM * AND SYNERGETICS *
32 REM *
34 REM *   ALL COMMERCIAL *
36 REM *   RIGHTS RESERVED *
38 REM *
40 REM *****

52 REM : COLOR KILLER DESCRIBED
54 REM : IN ENHANCING YOUR
56 REM : APPLE II, VOLUME I.

100 HOME=: REM  CLEAR SCREEN
110 HGR : REM  HIRES ON
120 POKE - 16301,0: REM  MIX GR
    APHICS

130 HCOLOR= 3: REM  DRAW WHITE

200 HPLOT 70,40 TO 120,40 TO 123
    ,110 TO 168,110 TO 171,40 TO
    221,40 TO 221,140 TO 70,140 TO
    70,40: REM  DRAW FIGURE

300 HCOLOR= 1:H = 90:V = 58: GOSUB
    500
310 HCOLOR= 2:H = 90:V = 115: GOSUB
    500
320 HCOLOR= 5:H = 190: GOSUB 500

```

PROGRAM 2-1, CONT'D...

```
330 HCOLOR= 7:V = 86: GOSUB 500
340 HCOLOR= 6:V = 58: GOSUB 500
350 HCOLOR= 3: HPlot 90,86 TO 99
    ,86 TO 99,96 TO 90,96 TO 90,
    87
360 REM    DRAWS SIX COLOR BLOCKS

400 VTAB 21: HTAB 12: PRINT "COL
    OR KILLER IS ";: REM    PRINT
    MESSAGE

410 POKE  - 16294,0: REM    TURN K
    ILLER OFF

420 NORMAL : HTAB 28: PRINT "OFF
    ";: REM    TEXT

430 FOR N = 0 TO 6000: NEXT : REM
    STALL 5 SECONDS

440 POKE  - 16293,0: REM    TURN K
    ILLER ON

450 INVERSE : HTAB 28: PRINT " O
    N ";: NORMAL : REM    TEXT

460 FOR N = 0 TO 6000: NEXT : REM
    STALL 5 SECONDS

465 IF PEEK ( - 16384) > 127 THEN
    POKE  - 16368,0: PRINT : PRINT
    "RUN MENU": REM    EXIT ON KP

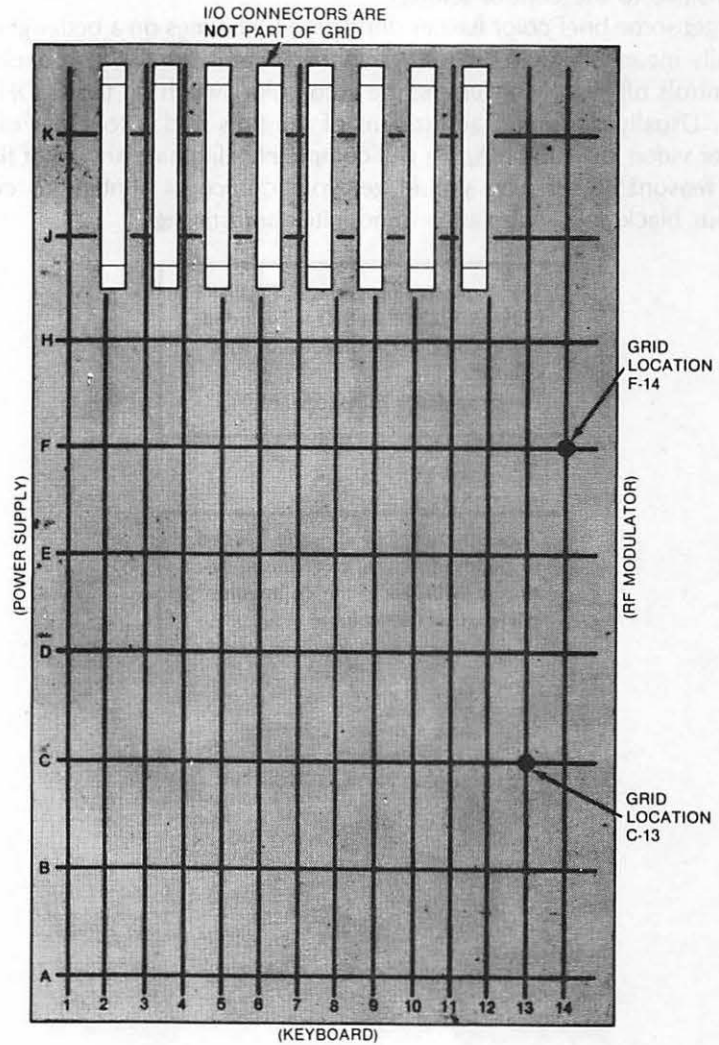
466 REM    REPLACE :PRINT: WITH
    :END IN 465 IF AUTO MENU IS
    NOT IN USE.

470 GOTO 410: REM    DO IT AGAIN

500 FOR N = 0 TO 9
510 HPlot H,(V + N) TO (H + 10),
    (V + N)
520 NEXT : RETURN : REM    SUB TO
    DRAW BOXES
```

APPLE BOARD PARTS LOCATIONS

Parts are located on the Apple main circuit board by use of a grid system



There are fourteen columns in the grid numbered from 1 to 14. Each numbered column goes from front to back. There are nine rows in the grid lettered from A to K. Each lettered row goes from left to right. Letters G and I are omitted to prevent mixups.

An integrated circuit, or IC, is located by finding the grid crossing that it is *nearest* to. If the IC is a big one, the *lowest* grid crossing number is used as the location.

For instance, the game paddle is located at J14. The 6502 microprocessor is said to be located at H6, although it also takes up grid locations H7, H8, and H9.

If the Apple is still in the case, you cannot see the grid numbers that run from left to right since these are at the front of the circuit board under the keyboard. Instead, use row "C" and count the fourteen integrated circuits from left to right. Integrated circuit C1 is a 74LS153, while integrated circuit C14 is a 74LS32.

Note that not all rows are completely filled, and that some ICs are bigger than others. Thus, the IC in grid location F14, which is a 74LS259 or a 9334, is only the tenth or eleventh *device* from the left, depending on the version of your Apple.

Note also that the row of numbers along the top of the board are the slot numbers for the I/O sockets and have nothing to do with the grid callouts.

REMEMBER—To find a given location, count GRID POSITIONS and **not** devices.

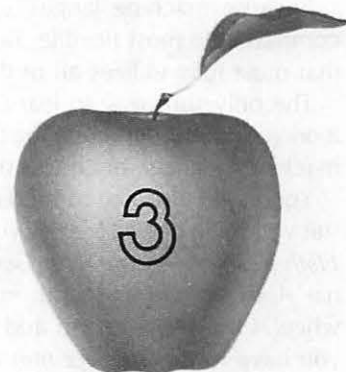
Our new software color killer only kills the color subcarrier at one point in the circuit, unlike Revision 7 and later Apple versions whose hardware color killer nails the subcarrier *twice*. Thus, the software color killer may be slightly more sensitive to the control settings.

If you get some brief color flashes during the OFF times on a better grade set, this usually means the auto-color circuitry in the tv is *hunting*. Try backing the color controls off slightly or turning the auto-color switch on the tv OFF if this happens. Usually, a careful adjustment of controls and a touchup of the rf modulator video level at the Apple will completely eliminate any color flashing. On any reasonable set, you should get your choice of continuous color or continuous black and white, after some initial adjusting 🍏

**The Applesoft program SOFTWARE
COLOR KILLER DEMO is included
in the companion diskette to this
volume.
The program is fully copyable.**

**A complete set of all parts needed
to build one color killer modifica-
tion is included in the companion
parts kit to this volume.**

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* are written 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 few shards that 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** be done 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 HRCC *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.

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.

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.

**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.

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!

Get yourself some scratch pads as well. Label the little blank one "Serendipity" and the big 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.

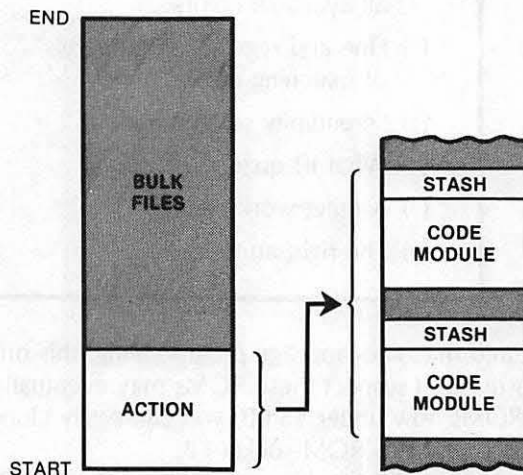


Fig. 3-1. A "typical" machine-language program.

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.

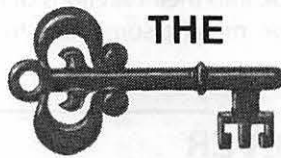
A stash is a short file that works directly with a module. The stash often follows immediately after the module that uses it. Typical stash entries might be a short ASCII string, a list of condition codes, or a table of indirect addresses. The stash holds values needed by the module that it works with.

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.

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 unless they are swapping things back and forth to the disk. 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

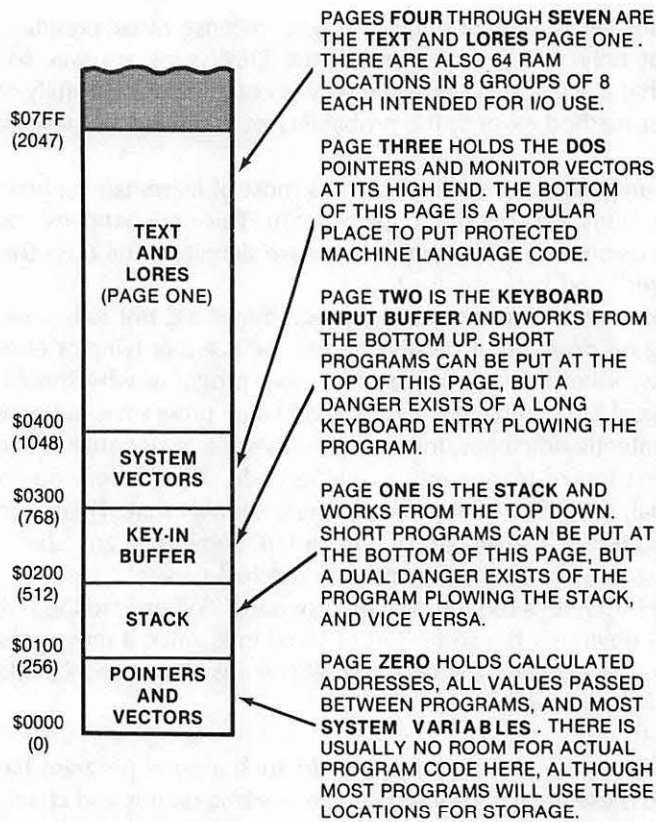


Fig. 3-2. Low RAM memory map.

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 best 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.

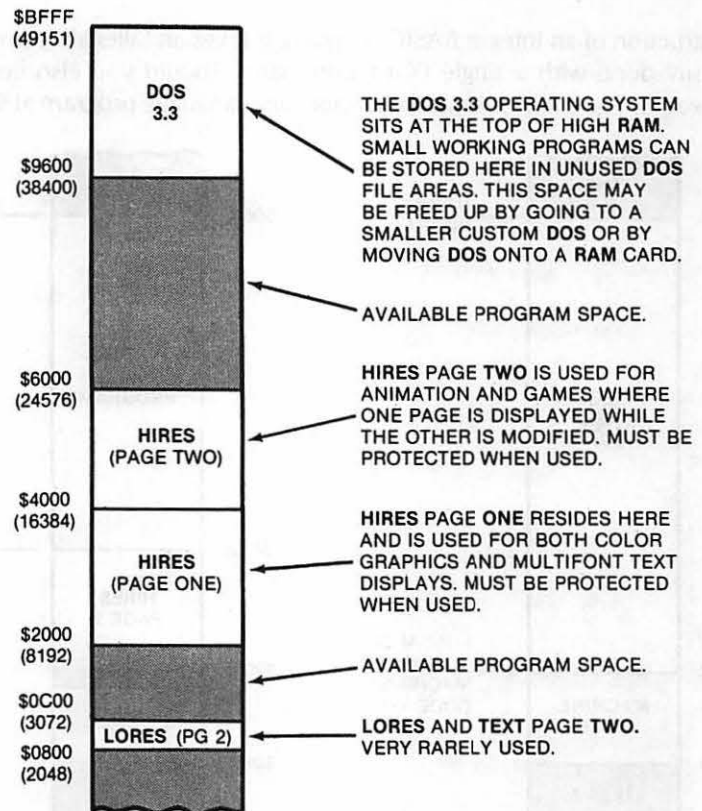


Fig. 3-3. High RAM memory map.

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 jump above HIRES and continue as far as they have to.

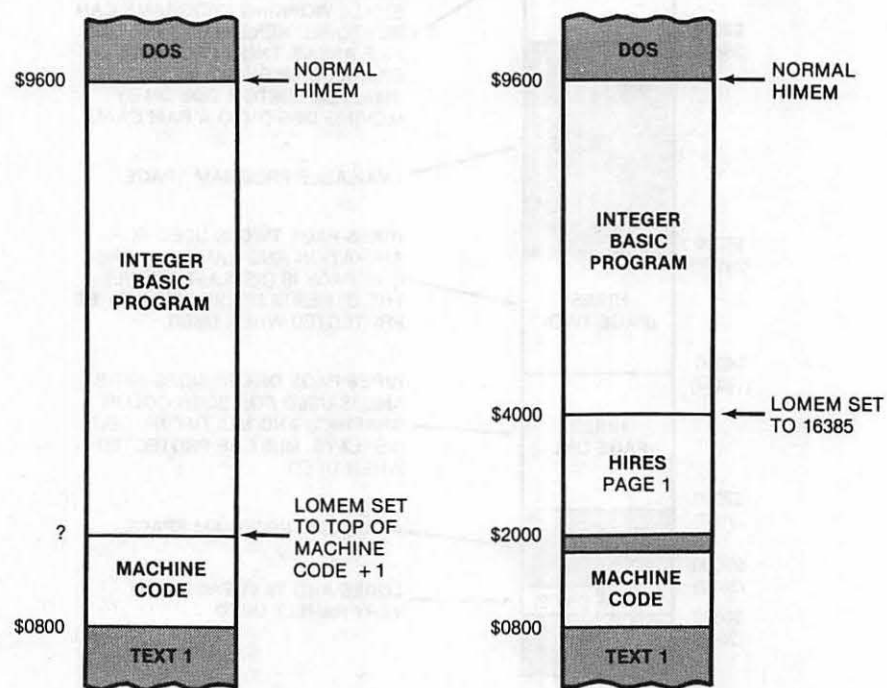
Combining programs

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.

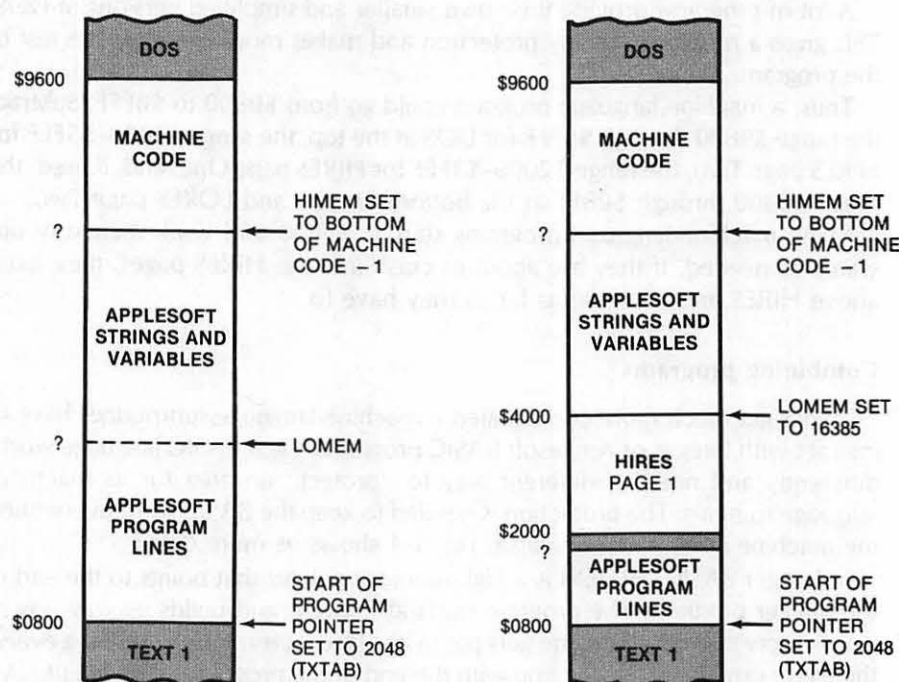
The usual way to tie a machine-language program into Integer BASIC is to start the machine-language sequence at \$0800 and set LOMEM to at least one space above the end of the machine-language code. This LOMEM can be set as the

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



(A) Integer BASIC, no HIRES.

(B) Integer BASIC using HIRES 1.



(C) Applesoft BASIC, no HIRES.

(D) Applesoft BASIC using HIRES 1.

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 $2K \times 8$ byte-wide 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.

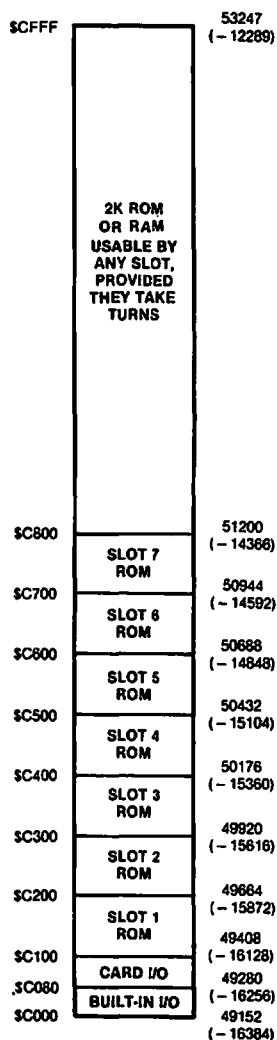


Fig. 3-5. I/O map.

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 single step and mini-assembler code, along with the old floating-point package. None of these machine-language test and debug features are available in the Autostart ROM.

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{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 or giving you extra RAM space.

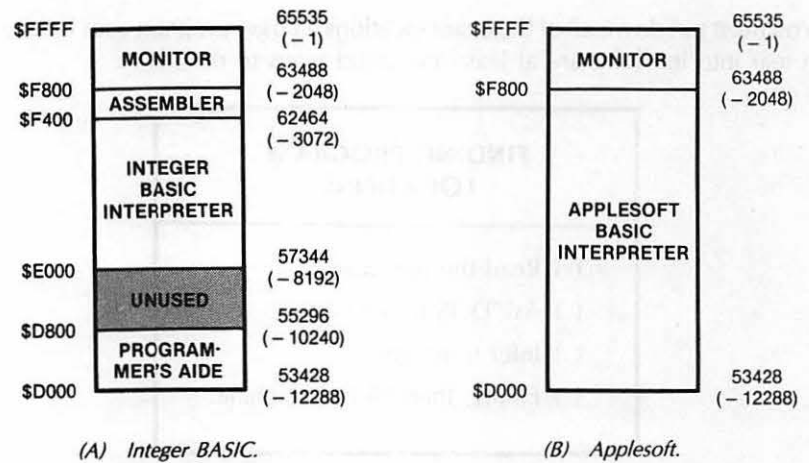


Fig. 3-6. High ROM maps. A plug-in ROM or RAM card can deactivate these and substitute its own code.

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
<ul style="list-style-type: none">() 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 DOS 3.3, the starting address ends up stashed 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.

The third method of finding a machine-language target program's lair is to infer where the program sits from what it has to do and what it has to interact with. Our HRCC gives us a good example here. We can't directly find where HRCC sits since it is an "R", or relocatable, rather than a "B", or binary file.

But, the Applesoft Toolkit book tells us HRCC 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 LOADHRCC that comes with the HRCC 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, HRCC sits from \$8DFF through \$95FF. This will include the HRCC 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 LOADHRCC 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.

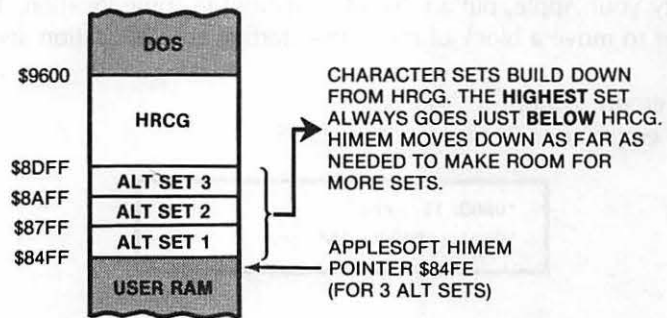


Fig. 3-7. Location of HRCG program and alternate character sets in a 48K Apple II.

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-booted 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. 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 HRCCG, 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 of a file.

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 book 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.

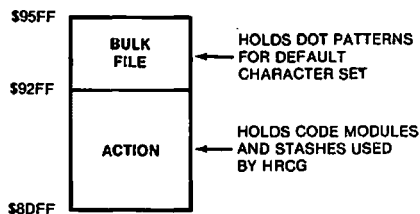


Fig. 3-8. Separating the action from the bulk files.

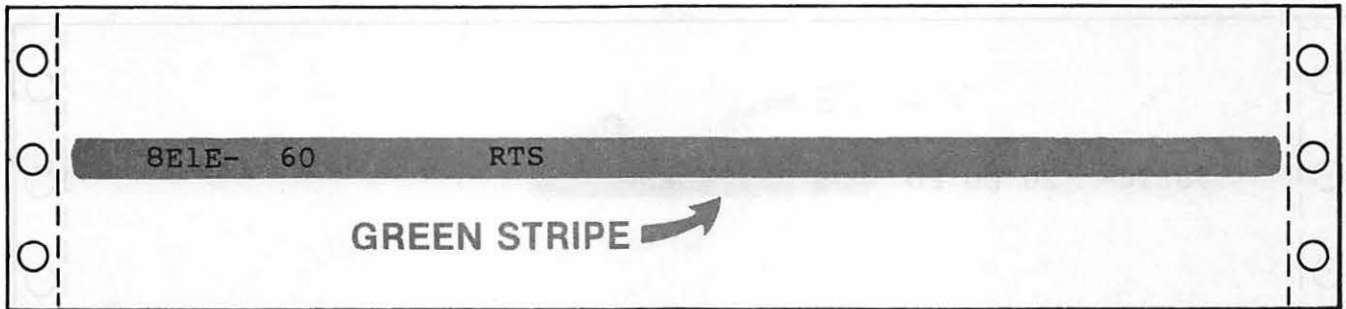
Don't worry, just yet, about the extra question marks we have above the bulk file. Somehow, these look "different" from the code in the bulk file. As you gain practice, these slight differences will leap out at you. But, our goal, here and

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 Achilles' 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 highlighter to put a green bar through the code. 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 highlighter!

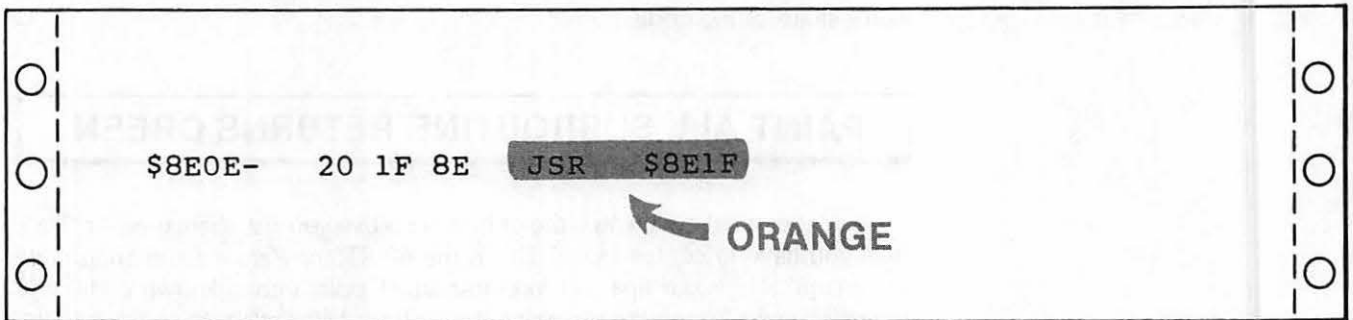
Now, things start to get interesting . . .

PAINT ALL SUBROUTINE CALLS ORANGE

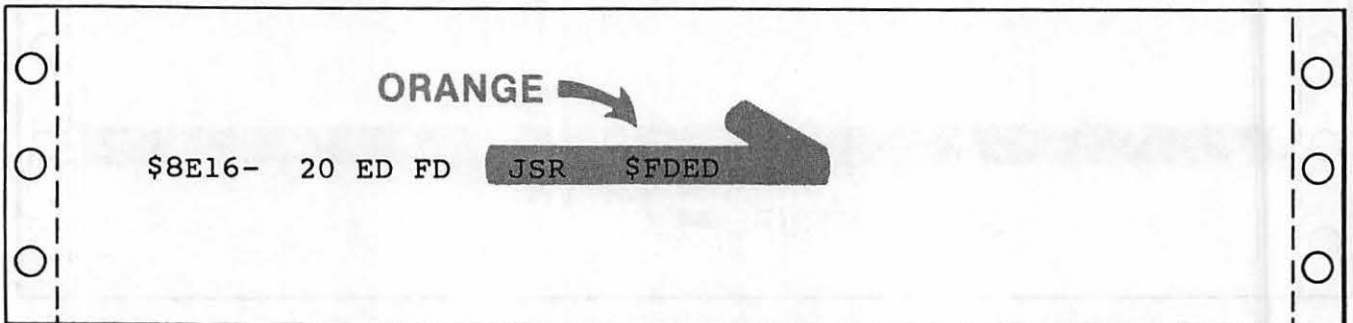
Next, get yourself an orange page highlighter and go through the action. Identify every rational JSR and its address in orange.

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 . . .



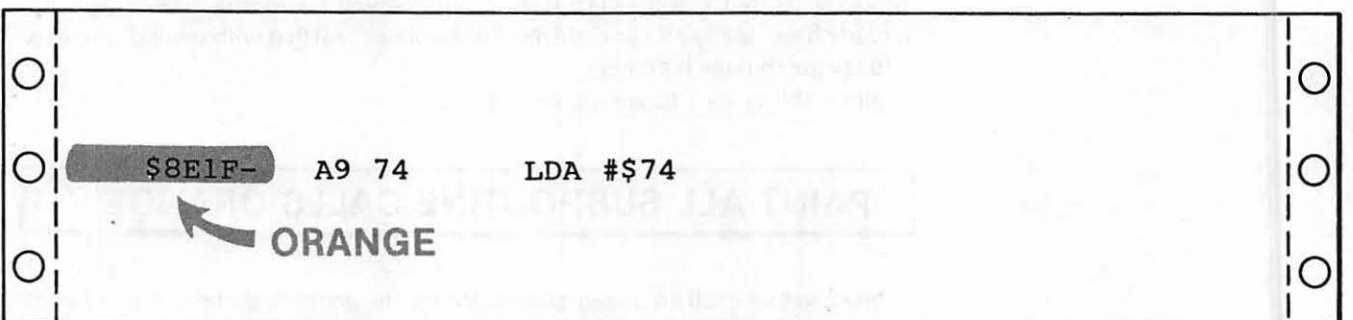
This subroutine call is in range, so we color only the JSR and the `$8E1F`.
For an out-of-range or "long distance" call, do it like this . . .



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 a 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 located 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.

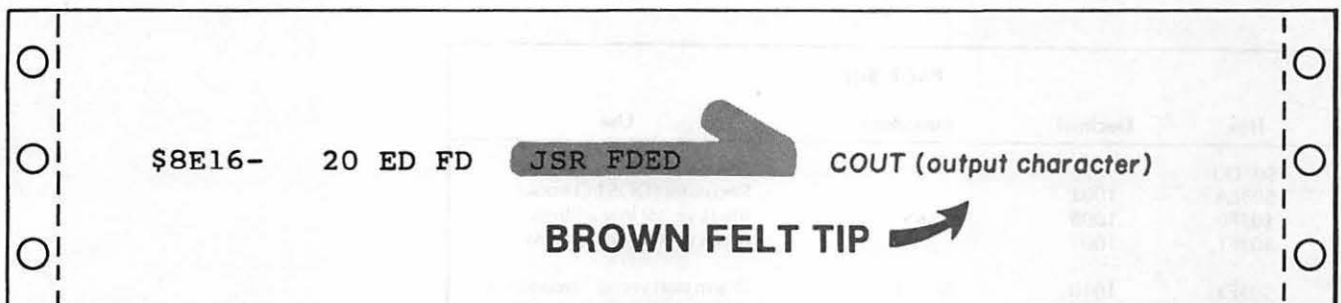
If you get a “JSR to the middle of . . .”, try relisting *from the JSR address* to see if you get rational code. This will help clarify the boundaries between stashes and code modules. We will see an example of this later.

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	BRKV	Re-enter DOS
\$03EA	1002		Reconnect DOS I/O hooks
\$03F0	1008		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 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	TAPEIN	Annunciator 3 ON soft switch
\$C060	- 16288		Cassette tape read input
\$C061	- 16287		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	HLIN	Draw a horizontal LORES line
\$F828	-2008	VLIN	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 program available. Eventually, you will want this list in numeric order, but for now, just list addresses as you run across them.

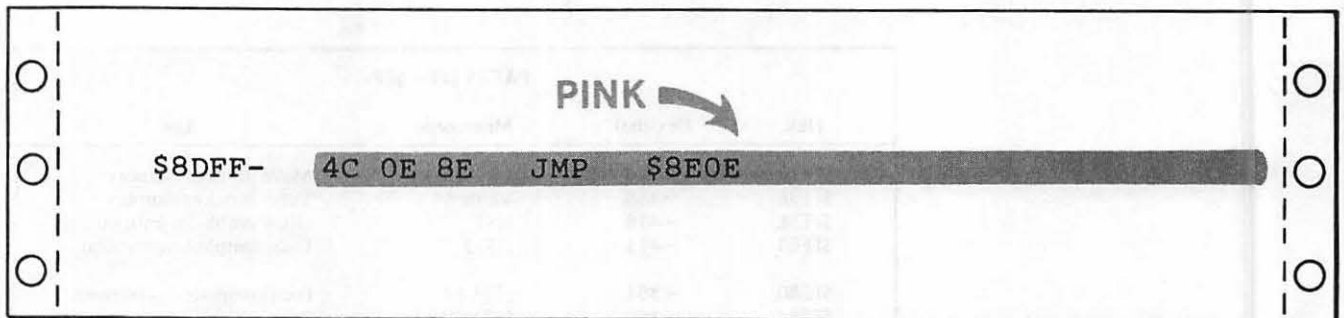
PAINT ALL ABSOLUTE JUMPS PINK

Ready for a new color? Get the pink highlighter 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 address 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.

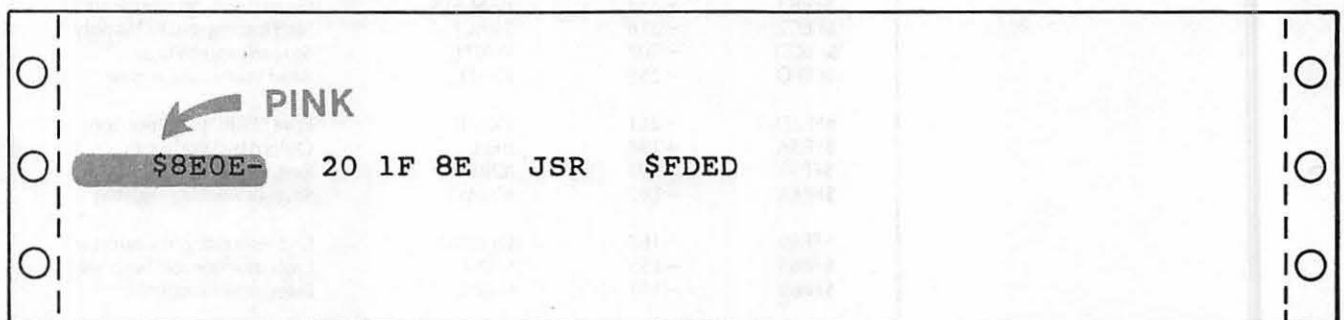
If the jumps are inside the action, then also put a pink line showing where the jump hopped to, just like you did with the subroutines. The jumper and jumpee may be connected vertically along the left-hand edge, but do this only if the two are less than twenty lines apart. Also "vote" on the most popular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

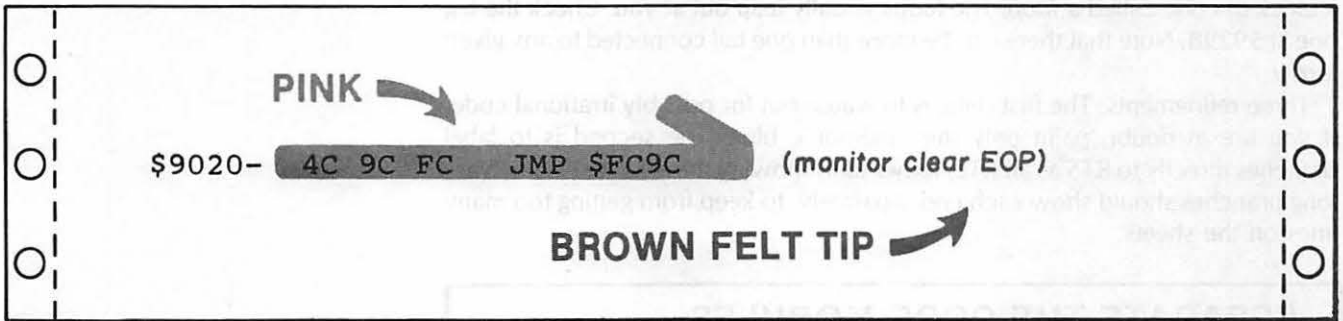
Here are the two steps that are involved in pinning down an inside-the-action jump . . .



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 enough ready to tear into the code.

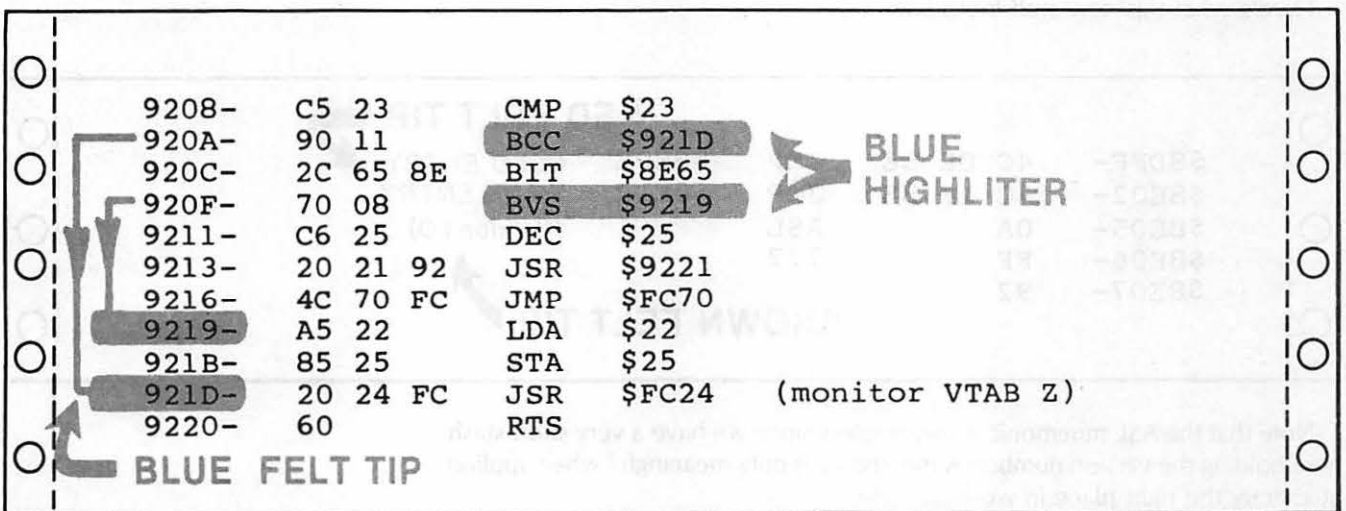
On an indirect jump using the (\$6C) code, go to the address shown in parentheses and identify this as an indirect address, and show the location that is using it for the indirect jump.

Let's hack away at our structure some more.

SHOW THE BRANCHES IN BLUE

Get out the blue page highlighter 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 blue felt-tip pen. Show the direction of each branch, and keep any branch lines from crossing.

Here is an example . . .



*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 "0R" 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 "0A" here, which apparently stands for version 1.0.

The "R" mentioned above may be new to you. The "R" means "relative" and is used with *relocatable* programs. "0R" 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 . . .

RED FELT TIP ↗					
○	\$8DFF-	4C 0E 8E	JMP	\$8E0E	COLD ENTRY
○	\$8E02-	4C 1F 8E	JMP	\$8E1F	WARM ENTRY
○	\$8E05-	0A	ASL		(Version 1.0)
	\$8E06-	FF	???		
○	\$8E07-	92			
BROWN FELT TIP ↗					

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 picture 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.

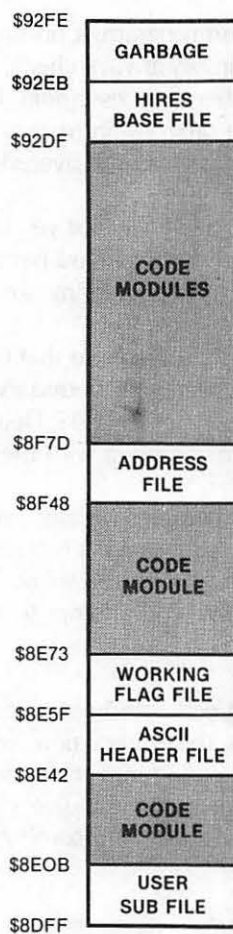


Fig. 3-9. Separating HRCG stashes from code modules.

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 on. The filter is valid if its answer leaps out at you and is 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
<ul style="list-style-type: none">() 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 HRCC 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.

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 HRCC, 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 basic byte patterns are distinctive, starting with a line number, followed by 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 LOADHRCC 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

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

We'll 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. HRCC 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 HRCC 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 HRCC, 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 HRCC, 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 HRCC 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. However, 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.

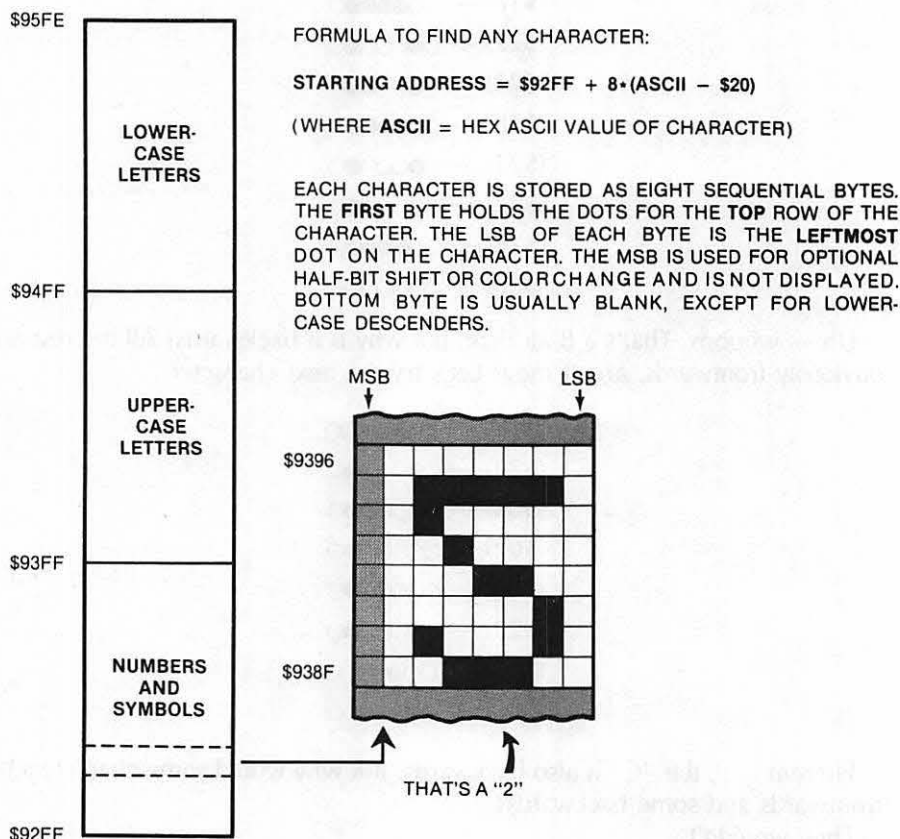


Fig. 3-10. How the HRCG default character set is stored in the bulk file.

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."

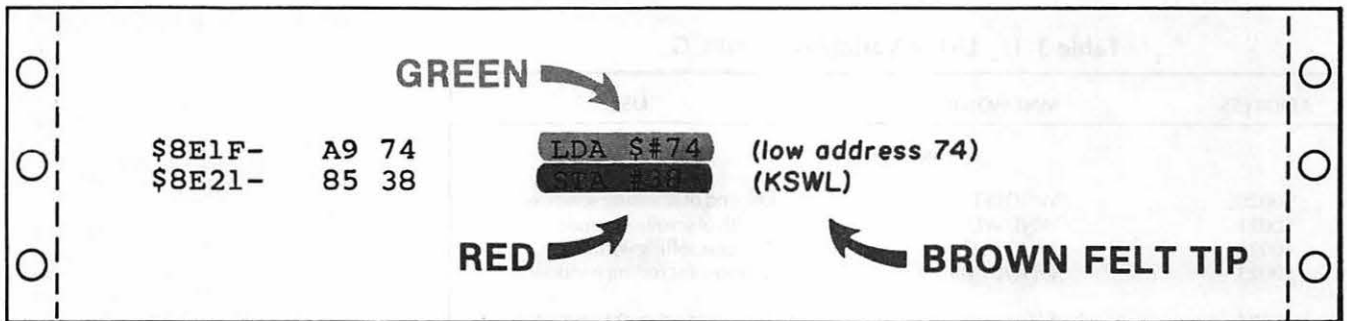
Now, go through the code modules line by line, and each time you find an address used for loading, storing, BIT testing, logic operations, or whatever, paint the variables pink and the constants green.

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	WNBDM	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	CLEOP	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	} YELLOW HIGHLIGHTER
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.

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 $\times 10$.
\$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 —
	<ul style="list-style-type: none"> \$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 WNDWIDTH 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 WNDWIDTH 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 WNDWIDTH 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 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.

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, 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 out 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 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 decipherer. 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.

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! 🍏

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.

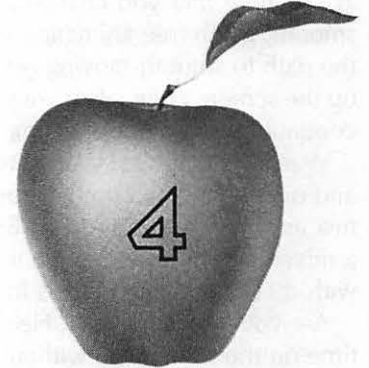
You can expect the "lister" to lie about one-half of the time when it comes out of a file or dead code and starts into legal code.

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.

Enhancement



FIELD SYNC

A one-wire hardware modification that makes for stunning new animation and graphics action effects. Now, you can exactly lock to video timing for split screens, high-accuracy light pens, and much more.

FIELD SYNC

This enhancement is so horrendously powerful that it's scary.

It is a one-wire plug-in modification to your Apple that gives you *field sync* and opens up so many mind-blowing new possibilities that we can't even begin to hint at them here. What field sync does is lock your display to your program. You can now flip soft switches, update display screens, or change pages while the screen is on its vertical blanking return trip. You can also keep ahead of or behind the live scan lines to keep your display from mixing "old" and "new" stuff at the same instant.

And, we will show you a brand new and incredibly powerful software method to lock to your video field timing that is **exact** to a fraction of a microsecond. This exact sync lets you mix and match display modes, both dynamically and "on the fly" for splits or wipes in any direction. Exact sync also can eliminate

90% of the parts and the hassle needed for a fast and precision light pen or for a touch screen.

So, what else is field sync good for?

For openers, gone forever are any glitches, flashes, and garbage that whip by at any time that you change an Apple screen mode. Field sync is the key to smooth, glitch-free animation without any collision or dropout effects. Here is the path to smooth-moving gentle scrolls that let you read things as they move up the screen. Here, also, are spinners and other action animation made totally continuous and smooth acting.

Want 3-D graphics? Want lots of different single-line colors in HIRES that cross and overlap in any combination? Want to mix LORES and HIRES? Or, want to mix and match text and LORES all over the screen? Or, all three? How about a mixed graphics mode with the text at the top? Or, in the middle? Or, graphs with quickly changed labels in both the X and Y direction?

Are you into animation? Field sync lets you compute and display at the same time on the same page without any glitches, dropouts, sugar, or other hassles.

What about some video wipes that smoothly move one page off the screen to reveal a second hidden page, done in any direction, at any speed, in any combination? Or, a choice of hundreds of LORES colors? How about a bomber flyby or a road race, smooth, fast, and alias-free?

Need some grey scale? Sure thing. How about some game symbols that mix a pair of ordinary text characters into a single symbol? Text over color? Naturally.

Maybe you would like a full color HIRES adventure where the text optionally "floats" in front of the graphics display. This is trivial with field sync.

All these only scratch the surface. We haven't even begun. And, it's real spooky to even think about what will happen when a really good programmer gets his hands on this mod.

And, best of all, it is back to the drawing board for you *Atari* people. Apple wins this round of "Whose-got-the-best-color-graphics?" bytes down.

In this enhancement, we will look at what a field sync modification is and how you can make one. We will also give you some very simple yet powerful support software. We'll then spend much of the rest of the book exploring a few of the more blatantly obvious uses of field sync.

Total cost of the enhancement is around \$2.00 and can be done in a few minutes. The modification will not void your Apple warranty and is completely removable. The support software, even that needed for an exact lock, takes only a few machine-language bytes and is easily reached from any language.

WHY SYNCHRONIZE?

Every personal computer designer faces the dilemma of how to compute and display video at the same time. The timing and video signals sent to your monitor or color tv are extremely critical and must be there all the time. Even the briefest mixup or delay, and you end up with a twisted, torn, or missing picture.

Apple solved this dilemma in a brilliant way. Check into the 6502 microprocessor timing and its memory-access activities, and you find out that the CPU only accesses memory on one half of each of its clock cycles. The Apple's clock cycle is slightly under one microsecond. Of this time, the 6502 microprocessor used as the Apple's CPU must have unrestricted memory access for only half of each microsecond.

On the other half of each clock cycle, the 6502 CPU and the rest of the Apple's computing circuitry couldn't care less what the main RAM memory was up to.

So, an elaborate hardware circuit called a *multiplexer* was set up that gives memory access to the Apple's computing circuitry for one half the time and gives the Apple's video-display circuitry memory access for the other half. Each is happy with its half of the access, and each piece of circuitry gets its access each and every microsecond.

The result is glitch-free and flicker-free video that is totally independent of any computing activity. Most importantly, this is done using the main memory rather than a separate video display memory. This means that no time or effort is needed to get between a result and putting that result on the screen. No wait for video memory access is ever needed. Nor is any time ever taken away from the critical video display waveforms. The operation is fully transparent and totally invisible.

Both the thinking and the design that went into the Apple's video timing circuitry was brilliant; it is a totally independent video that is immediately accessible to all.

But, they overdid it.

The video display timing is completely independent from the computing timing, although both are derived from common signals. This means that there is no immediate way for the computing circuitry to tell where the video-display circuitry happens to be in its timing cycle.

Field sync gets around this total independence by taking a sample waveform from the video display timing and routing it back into a location where you can test it with software. We will use part of the cassette read circuitry as an input to feed back this timing signal for us.

Let's take a closer look at the Apple's video timing and see if we can't find a good waveform to give us field sync.

Timing waveforms

A simplified block diagram of the Apple's timing chain is shown in Fig. 4-1. There are three main parts to this chain. These are the *video rate* timing, the *horizontal rate* timing, and the vertical or *field rate* timing.

The video-rate timing is the fastest. It starts with a 14.318-megahertz crystal oscillator that is the master timing reference for everything in your Apple. This master reference is divided by four to get the standard color subcarrier frequency of 3.579 megahertz. The master reference is further divided by 3-1/2 to get a CPU clock frequency of 1.023 megahertz.

This CPU clock frequency also sets one horizontal character time on the screen. The horizontal character time is equal to seven video dots. In HIRES, these seven dots equal the least significant seven bits in the data word, arranged backwards. In standard text, these seven dots consist of two blank undots and five horizontal dots from the output of a dot matrix character generator. The character generator receives the bits in the data word and then converts them to the proper dot patterns for you.

Either way, seven dots go on the screen in one CPU clock cycle. These seven dots equal three and one-half color clock cycles of 3.58 megahertz each. The color of the dot will be decided by the position or the time delay of the dot with respect to the *phase* of the color-reference signal.

The video rate timing does several other things for us. The two clock phases needed by the CPU are split out with this timing. The video timing automatically gets the seven dots lined up in the right place in the right time for us. And it automatically takes care of the multiplexing needed so that the memory can be shared between the display and the CPU. This memory multiplexing is more complicated than it would seem at first since the memory chips used make each

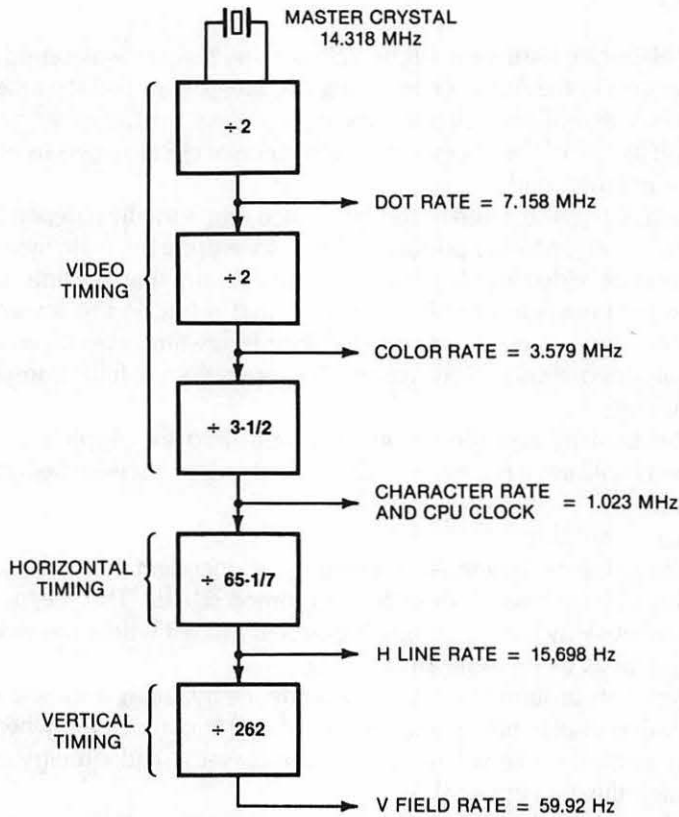


Fig. 4-1. Details of Apple's timing chain.

RAM address pin do double duty. Thus, the multiplexer has to go through *four* steps each clock cycle and not just two.

The CPU clock frequency of 1.023 megahertz is the lowest frequency needed by the computing side of the Apple. All of the rest of the timing chain is mainly involved in video display timing. Outside of their sharing the same high-frequency timing, the computing side and the display side of the Apple are more or less independent.

The horizontal rate timing sets the line length and the horizontal sync rate for us. There are 65 possible character positions in a horizontal line. Of these, 40 are "live" video slots and 25 are "blank" video slots. A live video slot can hold seven HIRES dots, a horizontal portion of a 5×7 dot matrix character, or a color pattern needed as part of a LORES color block.

Fig. 4-2 shows us the waveform of a single horizontal line. We start with the blank character positions. Into these blank character positions, we put a "blacker

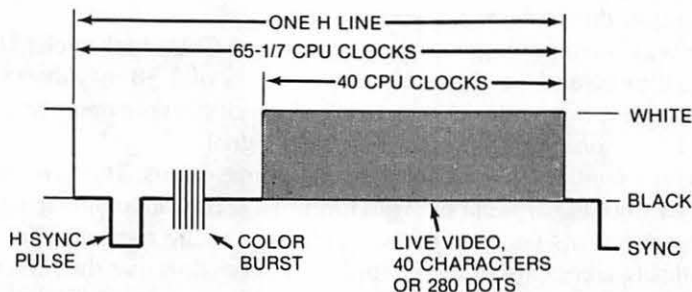


Fig. 4-2. Horizontal rate timing display is black or off screen except for live video time.

than black" horizontal sync pulse, followed by a reference burst of a number of 3.58-megahertz color-reference cycles. The exact number and position of these reference cycles depend on the version of your Apple. Some older Apples would not give you color on certain video recorders or on a few oddball brands of premium tv sets. The Revision 7 upgrade seems to have eased this.

After the horizontal sync pulse and the color burst, we have some more blank character locations. These are followed by the 40 live characters per line.

There is one tiny but crucial detail added in the horizontal timing. The line is an odd number of CPU clock cycles in length. Each clock cycle equals 3.5 color cycles, which means that when you get to the start of the next line, the color reference will be half a cycle off. To get around this, the video timing is delayed by half a color clock cycle *once each horizontal line*. Thus, every sixty-fifth CPU clock cycle will be wider by half a color cycle, or around 140 nanoseconds wider than usual.

This handles a key difference between commercial color-tv broadcasts and Apple signals nicely. In commercial color tv, you want the color subcarrier to cancel each successive horizontal line so that the subcarrier doesn't interfere with the luminance video. In the Apple, though, the subcarrier *is* the video, so you want its phase to be the same on each line. If you didn't do this, a green dot on one line would be a violet one immediately below, and so on.

At any rate, our horizontal line consists of 25 blank locations followed by 40 live locations. One of these blank locations is slightly wider than the others to keep the colors in step. Our horizontal sync frequency ends up at 15,698 hertz. This compares with the 15,735 hertz of a color commercial tv broadcast or the 15,750 hertz used for black and white commercial tv broadcasts.

The horizontal-rate timing also is used to refresh the dynamic RAMs. The timing is arranged to exercise the RAMs in such a way that they continue to hold valid data for us. This refresh is invisible and "free" since it automatically takes place as part of the normal display timing.

The vertical-rate timing takes the horizontal timing and divides it down by a factor of 262, giving us the vertical waveforms shown in Fig. 4-3. There are 192 live lines and 70 blank lines. The 192 live lines are sequentially scanned in HIRES. In LORES, the 192 lines are clustered into 48 groups of 4 lines each, one for each LORES block on the screen. In TEXT, the 192 lines are arranged into 24 groups of 8 lines each, one for each of the possible vertical dot positions needed in a 5×7 plus blank dot matrix character.

A vertical sync pulse is provided in the middle of the vertical blanking time. This vertical sync pulse is much wider than a horizontal sync pulse. This lets the tv separate its vertical sync locking command from its horizontal sync locking commands.

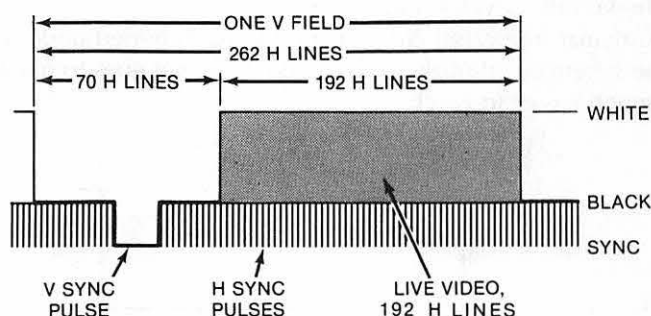


Fig. 4-3. Vertical-rate timing waveforms. Any mapping or soft switching done during blanking or retrace will be invisible.

Since there are 65 CPU clock cycles per line and 262 lines total, there are apparently $65 * 262 = 17,030$ clock cycles per field. The field frequency is 59.92 hertz. This compares with the 60 hertz of a black and white commercial broadcast or the 59.94-hertz vertical rate of a color broadcast.

Let's sum up some of these magic numbers . . .

APPLE SYSTEM CONSTANTS
The CPU clock frequency is 1.023 megahertz and has a time width of 0.978 microsecond. Every sixty-fifth clock pulse is slightly longer.
A horizontal line takes 65 CPU clock cycles and consists of 25 blank and 40 live cells. Each cell holds seven HIRES dots or one row of dots from a dot matrix character.
Horizontal frequency is 15,698 hertz.
A vertical field takes 262 horizontal lines and consists of 70 blank and 192 live scans. Live scans are used by eights for text, by fours for LORES, and by ones for HIRES.
Field frequency is 59.92 hertz.
There are 17,030 CPU cycles per field.

The timing waveform we want for field sync will have a rising edge at the beginning of the vertical blanking time and a falling edge at the start of the next field. If we look at the timing waveforms immediately off the timing chain, we see that this waveform is not available. Waveform V5 could be used, but it is only six horizontal lines wide, and, also, it is backwards from what we really want.

Instead, there is a vertical blanking waveform derived by an AND gate in B11. The magic signal outputs on pin 8 of B11. (See the Apple board parts locations sidebar at the end of Enhancement 2 for details on the Apple's parts locations.) We've shown this waveform in Fig. 4-4.

This particular integrated circuit is pretty much buried under the keyboard. Check the schematic, though, and we find its output goes to pin 4 of C14. This point is much easier to reach.

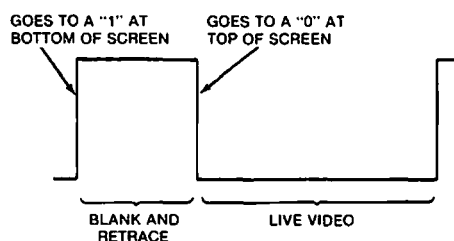
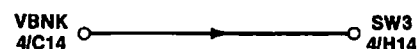


Fig. 4-4. Vertical blanking waveform used for field sync appears on pin 4 of C14.

Fig. 4-5 shows us the "schematic" of the field sync modification. We connect the vertical blanking waveform to a point in the cassette receiver circuitry that sometimes sees use as a "phantom" fourth pushbutton input. Electrically, this means that we jumper pin 4 on C14 to pin 4 on H14.

Fig. 4-5. "Schematic" of the field sync modification. One wire does it.



Our vertical blanking waveform is low for the 192 live scan lines and high for the 70 blank scan lines used for vertical retrace. If we can find the leading edge of this waveform with some simple software, we will have the entire blanking time to do things to the screen that will not show up till the next field. Also, if we can find the trailing edge, we can find out exactly when the next field will begin.

Many of the uses of field sync are just to simply flip a soft switch or two at any old time during the vertical blanking interval. Other uses may want to use every available microsecond of the vertical blanking time to remap an animated sequence. Yet others will need to find the exact start of a field for mixed field displays, or for use with either a precision light pen or a touch screen.

Building it

The field sync modification consists of a pair of sockets with a jumper wire between them. The mod plugs into the Apple main board. You can easily remove the modification later if the unit must be sent in for warranty repairs or whatever.

The modification also has to be removed if you use the cassette playback circuitry or if you use an external game paddle add-on that needs a "fourth" push-button input SW3.

These are the parts you will need . . .

PARTS LIST FOR FIELD SYNC MODIFICATION

- 1 — 16-contact quality DIP socket, machined-pin style.
- 1 — 14-contact quality DIP socket, machined-pin style.
- 1 — piece of No. 24 solid wire, insulated, 7-1/2 inches long.
- 1 — short piece of electronic solder.

And, here are the tools you will need . . .

TOOLS NEEDED TO MAKE THE FIELD SYNC MODIFICATION
() Needle nose pliers
() Wire stripper
() Small soldering iron, 35 watt
() IC puller (optional)
() Any old 14- or 16-pin integrated circuit
() Small vise or clamp

Fig. 4-6 gives us the complete construction details. Be absolutely sure to use high-quality machined-pin style DIP sockets here, since these are the only kind that can easily and safely be plugged into another DIP socket.

The "dummy" integrated circuit you plugged into the socket helps prevent the plastic from melting and keeps the pins aligned but you still have to be careful. When soldering to these sockets, be sure to prevent any shorts between adjacent pins 3 and 5, and be careful not to heat soften and distort either socket.

A pictorial of the field sync modification is shown in Fig. 4-7. If you decide to make your sync mod more or less permanent, you may want to tack the wire to the board with hot glue, silicon rubber, or some other "semi-permanent" gunk. Be careful not to get any glop on the socket pins.

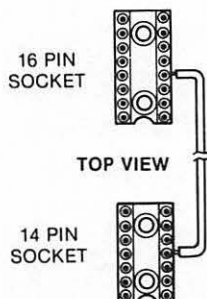
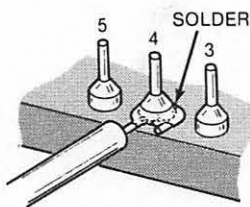
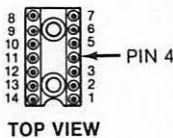
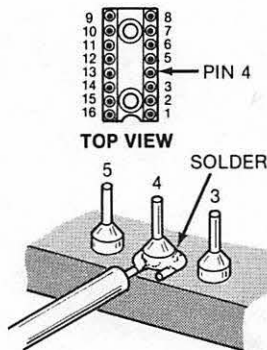
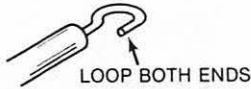
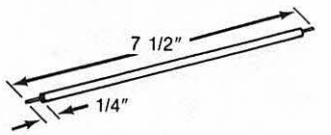
Should you ever want to remove your field sync modification, just reverse the process that is shown. While this modification does, in fact, void your Apple warranty, if you are very careful not to bend any pins or get any ICs mixed up, you should be able to remove any signs that the modification was ever in use.

SUPPORT SOFTWARE

Interpreted Applesoft is far too clumsy and way too slow to directly use field sync. You'll find field sync is best handled by short and fast machine-language subroutines. These subroutines can be called from any language you like.

Program 4-1 gives us six simple field sync utility subs, while Program 4-2 is an Applesoft routine that will test your field-switch modification for you and give you an alternating text and HIRES display.

The field-switch modification routes a copy of the vertical blanking waveform of Fig. 4-4 into the cassette circuitry at "SW3." SW3 is software tested in location hex \$C060. When you are in the *live* portion of the scan, a *zero* or *low level* gets routed to the most significant bit of \$C060, and a test of this location will show a *positive* number in 2's complement signed binary. When you are in the *blank* portion of the scan, a *one* or *high level* gets routed to the most significant bit of \$C060, and a test of this location will show a *negative* number here, again in 2's complement signed binary.



INSTRUCTIONS FOR BUILDING FIELD SYNC MODIFICATION

1. Cut a piece of insulated No. 24 solid wire to a length of 7½ inches. Then strip ¼ inch insulation off each end.

Form a tight loop in each end as shown.

2. Take a 16-pin machined-contact DIP socket and identify pin No. 4 by inking the plastic. Plug any old nonvaluable integrated circuit you have on hand into this socket. This will keep the pins aligned should the plastic soften. Secure this socket in a vise so you can work with it.

Note that this **MUST** be the type of premium socket that has small machined-pin contacts that are safe to plug into another socket.

3. Solder one end of the 7½-inch wire to pin No. 4 of the 16-pin DIP socket **EXACTLY** as shown. Be careful not to melt the plastic.

Be sure that no solder gets on the part of the pin that must fit into socket H14 on the Apple's main board and be sure that there is no short to adjacent pins 3 and 5.

4. Take the 14-pin machined-contact DIP socket and identify pin No. 4 by inking the plastic. Move the nonvaluable integrated circuit into this socket.

Secure this socket in a vise just like you did the earlier one.

5. Solder the remaining end of the 7½-inch wire to pin No. 4 of the 14-pin DIP socket **EXACTLY** as shown.

Again, be careful not to melt plastic, get solder on the pin, or short adjacent pins.

6. Arrange the field sync modification as shown. Note that the wire goes past pin No. 1 on the 16-pin DIP socket and past pin No. 7 on the 14-pin DIP socket.

7. If you want to, add a dab of silicon rubber, epoxy, or other glop to the wire where it leaves each socket body. This will act as a strain relief and keep the wire from breaking.

Remove the integrated circuit to complete your field sync modification. Put this IC away and out of sight. Refer to text for installation and checkout.

Fig. 4-6. How to build the field sync modification.

Here is how you install your field sync modification.

INSTALLING YOUR FIELD SYNC MODIFICATION

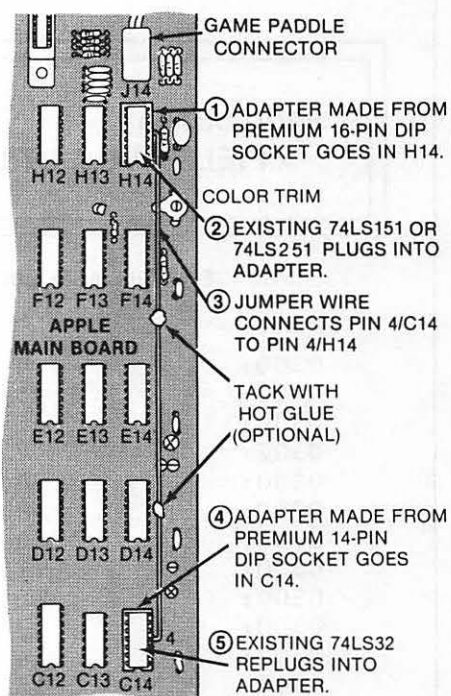
1. Turn off the Apple and unplug *both* ends of the line cord.
2. Verify that no other use is being made of the cassette read circuitry, such as a "fourth" push button SW3 add-on.
3. Remove integrated circuit C14, a 74LS32, using an IC puller if you have one.
4. Plug the 14-pin end of your field sync mod into C14. Make sure the wire is to the right and the notch is pointing towards the keyboard.
5. Plug the 74LS32 into the 14-pin socket now at C14. BE SURE NOTCH POINTS TO KEYBOARD!
6. Remove integrated circuit H14, a 74LS151 or 74LS251, using an IC puller, if you have one.
7. Plug the 16-pin DIP end of the field sync mod into H14. Be sure that the wire goes to the right and the notch points towards the keyboard.
8. Plug the 74LS151 or 74LS251 into the 16-pin socket now at H14. BE SURE NOTCH POINTS TO KEYBOARD!

A machine-language command called a BIT test will automatically transfer the MSB of \$C060 into the N flag for you, allowing easy branching. The only confusing part is that the *blank* time gives you a positive voltage and a 1 to the game connector which, in turn, is read as a *negative* number. The *live* scan time gives you a zero voltage and a 0 to the game connector which, in turn, is read as a *positive* number. This oddball turn of things is caused by the 2's complement signed binary used by the 6502's branch testing.

Anyway . . .

Your field sync is tested at location \$C060 by the command BIT \$C060, or "2C 60 CO".
This test **CLEARs** the N flag during live scan times and **SETs** the N flag during vertical blanking times.

Fig. 4-7. Pictorial shows field sync modification.



In Program 4-1, we will show you six simple ways to use your field sync. These are shown as program modules 1 through 6. These short machine-language modules are all separately located on memory page \$03. You can easily relocate them in any protected place in your machine that you want.

Examples of protected space in your Apple are the lower part of page 3 (when available), any locations set aside below LOMEM in Integer BASIC, any locations set aside above HIMEM in Applesoft, or one of the slots intended for optional character set that are used in HRCCG.

For simplicity, we will show all the examples starting at separate locations on page \$03. You can relocate these modules anywhere that they won't get plowed by something else.

Our first subroutine is the simplest, being all of six bytes long. It is called CRUDE and is just that. We enter this subroutine whenever we want to find the vertical blanking time. If we are in the vertical blanking time already, the BIT test will immediately give us a set N flag, and the BPL branch test will immediately fail, giving us a fast return. If we are in the live scan time, the BIT test will clear the N flag, and the branch will then try again. We keep trying, once each seven CPU cycles, till we finally reach the blanking time. Then we exit.

With this sub, you can always be sure you will return sometime during the vertical blanking interval. But, notice that you must not try recalling CRUDE more than once per field. If you have a fast machine-language program, calling CRUDE could return several times in the same field and really foul things up.

The big disadvantage of CRUDE is that you never know exactly how much time you have available before the next scan begins. Three times out of four, you will have the full blanking time. But every now and then, you will catch the blanking at the last possible instant. This only leaves you with a dozen or so useful microseconds before the next live scan starts.

For instance, say you happen to sample on the very last CPU clock cycle of the blank portion of the field. Two more CPU clock cycles will be taken up by the branch test, which fails, and six more CPU clock cycles will be needed for the RTS. This actually puts you into the first live scan line by 7 CPU clock cycles.

PROGRAM 4-1 **FIELD SYNC UTILITY SUBS**

LANGUAGE: APPLE ASSEMBLY

NEEDS: FIELD SYNC MOD

```

0300:      4 ; *****
0300:      5 ; *
0300:      6 ; *   FIELD SYNC   *
0300:      7 ; *   UTILITY SUBS  *
0300:      8 ; *   ($300.392)    *
0300:      9 ; *
0300:     10 ; *   VERSION  1.0   *
0300:     11 ; *   ( 7-15-81)    *
0300:     12 ; *
0300:     13 ; *   COPYRIGHT 1981 *
0300:     14 ; * BY DON LANCASTER *
0300:     15 ; *   AND SYNERGETICS *
0300:     16 ; *
0300:     17 ; *   ALL COMMERCIAL *
0300:     18 ; *   RIGHTS RESERVED *
0300:     19 ; *
0300:     20 ; *****

```

```

0300:     22 ; THESE SIX MACHINE LANGUAGE
0300:     23 ; MODULES ACCESS THE FIELD SYNC
0300:     24 ; MODIFICATION DESCRIBED IN
0300:     25 ; ENHANCEMENT #4 OF ENHANCING
0300:     26 ; YOUR APPLE, VOLUME I.

```

```

0300:     28 ; MODULES MAY BE RELOCATED IN
0300:     29 ; ANY PROTECTED SPACE.

```

```

FCA8:     31 DELAY   EQU   $FCA8      ; MONITOR DELAY SUB
C050:     32 GRAPHIC EQU   $C050      ; GRAPHICS SCREEN SWITCH
C000:     33 KEYBD   EQU   $C000      ; KEY PRESS CHECK
C063:     34 SYNC    EQU   $C060      ; VBLANK VIA "SW3"
C051:     35 TEXT    EQU   $C051      ; TEXT SCREEN SWITCH

```

PROGRAM 4-1, CONT'D...

```
0300:          38 ;  MODULE #1 -- "CRUDE" --
0300:          40 ;  THIS MODULE RETURNS SOMETIME DURING
0300:          41 ;  THE VERTICAL BLANKING INTERVAL.
0300:          43 ;  TO USE, JSR $0300 FROM MACHINE LANGUAGE
0300:          45 ;  DO NOT USE MORE THAN ONCE PER FIELD!
0300:          47 ;  THIS MODULE MAY BE USED EVERY FIELD.
0300:          48 ;  IT WILL HANG IF FS MOD IS ABSENT.
```

```
0300:2C 60 C0    50 CRUDE    BIT    SYNC    ; LOOK FOR BLANKING
0303:10 FB      51          BPL    CRUDE    ;  AND REPEAT TILL FOUND
0305:60         52          RTS          ; THEN EXIT
```

PROGRAM 4-1, CONT'D...

```
0310:          55          ORG  CRUDE+$10

0310:          57 ;  MODULE #2 -- "FEDGE" --

0310:          59 ;  THIS MODULE RETURNS AT START
0310:          60 ;  OF NEW FIELD WITH SLIGHT JITTER.

0310:          62 ;  TO USE, JSR $0310 FROM MACHINE LANGUAGE

0310:          64 ;  THIS MODULE MAY BE USED EVERY FIELD.
0310:          65 ;  IT WILL HANG IF FS MOD IS ABSENT.


0310:2C 60 CO  67 FEDGE  BIT  SYNC      ; LOOK FOR LIVE SCAN
0313:10 FB     68       BPL  FEDGE      ;   AND RETRY TILL BLANK
0315:2C 60 CO  69 BLANK  BIT  SYNC      ; LOOK FOR BLANKING
0318:30 FB     70       BMI  BLANK      ;   AND RETRY TILL LIVE
031A:60        71       RTS              ; THEN EXIT
```

PROGRAM 4-1, CONT'D...

0320: 74 ORG CRUDE+\$20

0320: 76 ; MODULE #3 -- "BEDGE"

0320: 78 ; THIS MODULE RETURNS AT START

0320: 79 ; OF VBLANK TIME WITH SLIGHT JITTER.

0320: 81 ; TO USE, JSR \$0320 FROM MACHINE LANGUAGE

0320: 83 ; THIS MODULE MAY BE USED EVERY FIELD.

0320: 84 ; IT WILL HANG IF FS MOD IS ABSENT.

0320:2C 60 C0	86	BEDGE	BIT	SYNC	; LOOK FOR BLANK
0323:30 FB	87		BMI	BEDGE	; AND REPEAT TILL LIVE
0325:2C 60 C0	88	FIELD	BIT	SYNC	; LOOK FOR LIVE SCAN
0328:10 FB	89		BPL	FIELD	; AND REPEAT TILL BLANK
032A:60	90		RTS		; THEN EXIT

PROGRAM 4-1, CONT'D...

```
0330:          93          ORG  CRUDE+$30

0330:          95 ;  MODULE #4 -- "ALTFLD" --

0330:          97 ;  THIS MODULE ALTERNATES BETWEEN
0330:          98 ;  TEXT AND GRAPHICS FIELDS.  IT
0330:          99 ;  EXITS ON ANY KEY PRESSED.

0330:         101 ;  TO USE, JSR $0330 FROM MACHINE LANGUAGE
0330:         102 ;  OR CALL 816 FROM EITHER BASIC

0330:         104 ;  THIS MODULE DISPLAYS CONTINUOUSLY
0330:         105 ;  TILL ANY KEY IS PRESSED.

0330:         107 ;  IT WILL HANG IF FS MOD IS ABSENT.


0330:20 10 03 109 ALTFLD JSR  FEDGE      ; FIND FIELD START
0333:8D 50 C0 110      STA  GRAPHIC    ; SWITCH GRAPHICS ON
0336:20 10 03 111      JSR  FEDGE      ; FIND FIELD START
0339:8D 51 C0 112      STA  TEXT       ; SWITCH TEXT ON
033C:2C 00 C0 113      BIT  KEYBD      ; HAS KEY BEEN PRESSED?
033F:10 EF 114      BPL  ALTFLD      ; CONTINUE IF NO KEY
0341:60 115      RTS                ; EXIT ON KEYDOWN
```


PROGRAM 4-1, CONT'D...

```
0350:          118          ORG  CRUDE+$50
```

```
0350:          120 ;  MODULE #5 -- "EXACTF" --
```

```
0350:          122 ;  THIS MODULE EXITS EXACTLY SEVEN
0350:          123 ;  MICROSECONDS INTO THE START OF A
0350:          124 ;  NEW FIELD WITH ZERO JITTER.
```

```
0350:          126 ;  UP TO SEVEN FIELDS MAY BE
0350:          127 ;  NEEDED TO ACQUIRE EXACT LOCK.
```

```
0350:          129 ;  TO USE, JSR $0340 FROM MACHINE LANGUAGE
```

```
0350:          131 ;  ONCE LOCKED, MAIN PROGRAM MUST
0350:          132 ;  CONTINUINE LOCK TILL DONE.
```

```
0350:          134 ;  THIS MODULE MAY NOT BE USED EACH FIELD.
0350:          135 ;  IT WILL HANG IF FS MOD IS ABSENT.
```

```
0350:2C 60 CO 137 EXACTF  BIT  SYNC      ; FIND BLANKING TIME
0353:10 FB    138      BPL  EXACTF      ;
0355:2C 60 CO 139 BLENK  BIT  SYNC      ; FIND FIELD START
0358:30 FB    140      BMI  BLENK       ; WITH JITTER
035A:EA      141      NOP              ; STALL FOR 2 CYCLES
035B:10 00    142      BPL  STALL       ; STALL FOR 3 CYCLES
035D:A9 3B    143 STALL  LDA  #$3B      ; DELAY FOR 17029
035F:20 A8 FC 144      JSR  DELAY      ; CLOCK CYCLES TOTAL
0362:A9 34    145      LDA  #$34      ; USING THE MONITOR
0364:20 A8 FC 146      JSR  DELAY      ; DELAY ROUTINE AND
0367:A9 01    147      LDA  #$01      ; THIS LOOP TIMING.
0369:20 A8 FC 148      JSR  DELAY      ;
036C:2C 60 CO 149      BIT  SYNC      ; HAVE WE BACKED TO START?
036F:10 EC    150      BPL  STALL      ; NO, GO BACK ONE MORE
0371:60      151      RTS              ; EXIT ON EXACT LOCK
```

PROGRAM 4-1, CONT'D...

```

0380:          154          ORG  CRUDE+$80

0380:          156 ;  MODULE #6 -- "EXACTB" --

0380:          158 ;  THIS MODULE EXITS EXACTLY SEVEN
0380:          159 ;  MICROSECONDS INTO THE BLANKING OF A
0380:          160 ;  NEW FIELD WITH ZERO JITTER.

0380:          162 ;  UP TO SEVEN FIELDS MAY BE
0380:          163 ;  NEEDED TO ACQUIRE EXACT LOCK.

0380:          165 ;  TO USE, JSR $0370 FROM MACHINE LANGUAGE

0380:          167 ;  ONCE LOCKED, MAIN PROGRAM MUST
0380:          168 ;  CONTINUE LOCK TILL DONE.

0380:          170 ;  THIS MODULE MAY NOT BE USED EACH FIELD.
0380:          171 ;  IT WILL HANG IF FS MOD IS ABSENT.


0380:2C 60 C0 173 EXACTB  BIT  SYNC      ; FIND LIVE SCAN TIME
0383:30 FB    174      BMI  EXACTB      ;
0385:2C 60 C0 175 BLUNK  BIT  SYNC      ; FIND BLANKING START
0388:10 FB    176      BPL  BLUNK      ; WITH JITTER
038A:EA      177      NOP              ; STALL FOR 2 CYCLES
038B:30 00    178      BMI  STAUL      ; STALL FOR 3 CYCLES
038D:A9 3B    179 STAUL  LDA  #$3B      ; DELAY FOR 17029
038F:20 A8 FC 180      JSR  DELAY      ; CLOCK CYCLES TOTAL
0392:A9 34    181      LDA  #$34      ; USING THE MONITOR
0394:20 A8 FC 182      JSR  DELAY      ; DELAY ROUTINE AND
0397:A9 01    183      LDA  #$01      ; THIS LOOP TIMING.
0399:20 A8 FC 184      JSR  DELAY      ;
039C:2C 60 C0 185      BIT  SYNC      ; HAVE WE BACKED TO START?
039F:10 EC    186      BPL  STAUL      ; NO, GO BACK ONE MORE
03A1:60      187      RTS              ; EXIT ON EXACT LOCK

```

*** SUCCESSFUL ASSEMBLY: NO ERRORS

But, each live scan line starts off with its 25 blank CPU clock cycles used for horizontal retrace, sync, and the color burst. The net time we have left is only $25 - 7 = 18$ CPU clock cycles.

Note that a clock cycle is slightly less than a microsecond. We could say "microsecond" when we really meant "CPU clock cycle" and wouldn't really be off that much.

At any rate, 18 CPU clock cycles is more than enough time to flip a few screen switches, but is far too short to do the extensive graphics remapping needed for animations or game graphics.

You use the CRUDE module as you would any old subroutine. Just do a JSR \$0300 from a machine-language program.

Note that most of these code modules will hang up on an unmodified Apple that does *not* have field sync installed. High-level user software should have some extra test included to make sure the field sync mod is in place before any of these modules are called. See page 116 for an example.

Module number 2 is named FEDGE and finds the start of a new field for us. What we do is wait till we are sure we are on the blanking part of the scan and, then, look for the instant that the live scan starts. This software tool is called an *edge detector*. If you happen to enter during the live scan time, the first BIT test spins its wheels till you reach the blanking time. Timing then drops through to the second BIT test which also spins its wheels till the start of the live scan time that tells us a new field has begun.

On the other hand, if you happen to enter during the blanking time, the first test fails immediately, and you drop through to the second BIT test that loops till the start of the next field.

The advantage of this FEDGE module is that it will always exit near the beginning of the live scan time. Since this module is edge sensitive, it always waits till the start of a new field before exiting.

Like CRUDE, module FEDGE can be used over and over again, once per field, as needed. After you exit FEDGE, you can go on and do anything you want with the machine.

There are several disadvantages to using FEDGE. The first disadvantage is that this routine will hang up on an unmodified Apple, since SW3 has to *change* to allow an exit. The second disadvantage is that there can be a jitter of up to 7 CPU clock cycles on the exit. This jitter is trivial compared to the jitter of CRUDE, but it is still not acceptable for precision field sync uses. Another disadvantage of FEDGE is that almost an entire field can be spent stuck in the subroutine if you aren't careful. This means you should adjust the rest of your program so that its length doesn't compound the time that it has to waste in the subroutine.

We can easily "inside out" our FEDGE sub to give us a BEDGE subroutine. This one is shown as module number 3 and exits on the start of blanking on the first available field. If you happen to enter during the blanking time, the first BIT test keeps retrying till the live scan time. Then, the second BIT test keeps retrying till you reach the start of the blanking time. Then you exit. You exit on the start of blanking plus one to seven CPU clock cycles. This leaves you with 70 full lines or some 4350 CPU cycles to do good stuff with before the start of the next field. This is more than enough time for simple animation where only a portion of the screen is to be remapped.

The BEDGE subroutine is best used when a medium amount of remapping is needed, while the FEDGE subroutine is best for flipping a single soft switch or two, or else, when used for remapping so extensive that you are using two HIRES pages and taking the entire field or more to update. Otherwise, the two subs have the same advantages and disadvantages.

Our fourth module is called ALTFLD and will automatically switch you from text on one field to graphics on the next. The module works by calling FEDGE, switching to graphics, calling FEDGE again, and, then, switching to text. It then looks for any key pressed and loops if there is no response. The display will mix text and HIRES for you as long as you like.

You can reach ALTFLD by a JSR to \$0330 from a machine-language program. Unlike our earlier modules, it is also feasible to directly reach ALTFLD from either BASIC. CALL 816 does it. We will see an example of this in Program 4-2.

This is a very simple field alternator. We may look at fancier field alternators in a future enhancement. We've included a simple field alternator here so that we have a way to test your field sync mod, and so that you can get started very simply and quickly on your own.

Module number 5 is the big surprise since many people would swear it is impossible to measure to an exact clock cycle with software which takes 7 CPU clock cycles to read and test at a port. These same disbelievers have gone out of their way to build all sorts of totally unneeded and ridiculous hardware so as to use things like precision light pens, touch screens, and so on. As we will see, it is trivially easy to quickly find *any* exact screen location using nothing but a few bytes of ordinary machine-language software.

We will look at two methods of doing an exact field lock. A simple software-only method appears here but a much faster and more flexible method that takes extra hardware will be shown you in Enhancement 13 of Volume 2.

Module EXACTF will find the *exact* start of a field for you, jitter free. To see how it works, we have to carefully look at machine-language instruction timing. The following procedure shows us the key secret.

The following steps illustrate how to do an EXACT screen lock using software. Remember that the last part of the FEDGE program looks like this . . .

BLANK	BIT SYNC
	BMI BLANK
	RTS

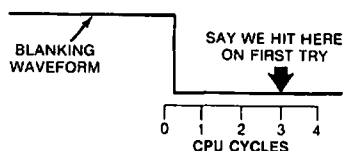
What this says is, "Read the port named SYNC. If the port is high, go and read the port again. If the port is low, exit immediately." This part of FEDGE is used to find the start of the new field for us.

But, the commands BIT and BMI take time. How much time? The BIT test takes 4 clock cycles and the BMI branch takes 3 clock cycles every time it repeats. This is a total of 7 clock cycles per try, or roughly 7 microseconds. Thus, it would seem that you can only read a port once every 7 clock cycles.

So, it would appear that there is an "inherent" 7-microsecond jitter in reading an outside random event that is routed to a port. And, indeed, there is if you read the port *only once*.

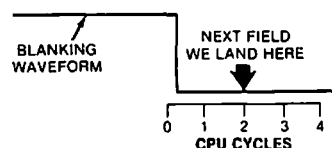
The preceding simple program will make its "last" measurement anywhere between one and seven microseconds after the port input goes low. Let's suppose that, this particular time around, we just happen to make our "last" measurement at 3 clock cycles into the start of the new field.

Like so . . .

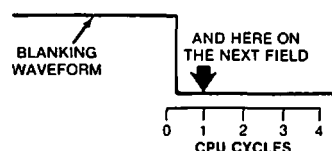


There are exactly 17,030 clock cycles in an entire field. Let's do a time delay of precisely 17,029 clock cycles and, then, make a new measurement. We are now only *two* microseconds into the next field, since going *forward* for 17,029 clock cycles is exactly the same as backing up *one* clock cycle on the next field.

This new measurement turns out this way . . .



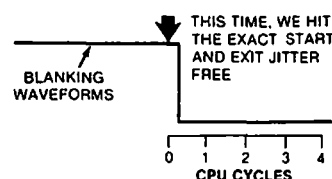
Now, delay another 17,029 clock cycles and we end up 1 microsecond into the next field, since we backed up one more clock cycle.



And, let's do it one final time. Only now, we "miss" the low part of the blanking waveform. But we see that we are EXACTLY at the start of the new field with zero jitter.

So, we exit.

We have done an exact lock to the blanking waveform

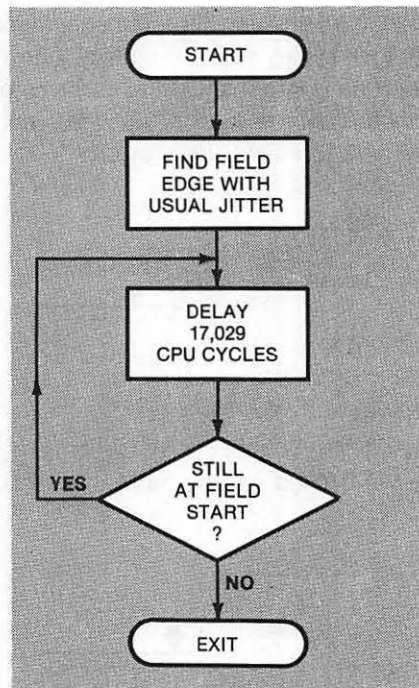


The number of fields till lock can be any amount from one to seven, but when you finally exit the final measurement, you have an EXACT lock on the last field that came up.

The logic behind a software EXACT lock says . . .

Find the start of a new field with the usual jitter.
Then, back up one clock cycle per field till you miss the start of the field.
Then exit.

Here's a flowchart that says the same thing



A BIT test of an absolute location takes 4 CPU clock cycles. A BPL or BNE branch needs 3 CPU clock cycles if taken. Thus, the fastest that we can possibly retest any absolute memory location seems to be once each 7 CPU clock cycles, or roughly once each 7 microseconds. Since you can enter these field sync modules at random, this implies that there is an inherent 7-microsecond jitter in any software measurement. And, 7 microseconds or 7 CPU clock cycles is 7 character slots across the screen. So, it looks like we are stuck with a 7-microsecond jitter, like it or not.

T'aint so.

Now, follow the bouncing ball. One entire field takes 17,030 CPU clock cycles, right? So, suppose we do a BIT test and, then, delay exactly 17,029 CPU clock cycles, or precisely and exactly 1 clock cycle *less* than one whole field. Regardless of where our field-start BIT test ends up, the new one takes place exactly 1 CPU clock *earlier* in the next field. *We have backed up precisely 1 CPU cycle.*

So, all you do is find the rough start of the next field using a code like FEDGE and, then, keep backing up *one cycle per field* till you miss the start. At that point, you exit. Your exit will take 2 CPU cycles for the test failed and 6 CPU cycles for the RTS, and you start at minus 1 CPU cycle from the field start. This exits you exactly 7 CPU cycles into the start of the new field. Remember that this leaves you with $25 - 7 = 18$ CPU cycles in the horizontal blanking time to play with. This is more than enough time to set up a split screen, a video wipe, or the timing for a light-pen measurement.

And, it is *exact*. There is *zero* jitter. We repeat. This will *exactly* find the start of a field for you.

There is a delay program called WAIT that is built into your Apple monitor starting at \$FCA8. We call this delay subroutine three times in a row to help us hit the exact timing that we need to delay 17,029 CPU clock cycles. Actually, we pick three magic delay values to total 17,016 CPU clock cycles. The remaining 13 CPU clock cycles are used up by the three immediate loads, the BIT test, and the branch in the loop.

What happens is that you find the leading edge of the new field using a FEDGE-like code, just like you did before. This gets done with 1 to 7 CPU clock cycles of jitter. Then, you back up exactly 1 CPU clock cycle per field till you hit the exact field start.

Then, you exit.

This is sort of like stopping your car suddenly at an intersection and, then, backing up to make sure you really were out of the crosswalk.

Exact sync does take a while. Up to seven fields may be needed to acquire an exact "lock." This will average out to around three fields or so per try, or something like one-twentieth of a second. But, this one-twentieth of a second is usually invisible since it is tacked onto the end of whatever it was that happened before.

Note that in light-pen uses, we may have to repeat the backing-up dose again when we sense the pen input. Thus, an average of a tenth of a second will be needed to hit a light-pen position of one of 40 exact character locations in one of 192 exact lines. You can thus easily find any one of 7680 screen locations in a tenth of a second by using nothing but a short piece of wire and some pure software.

A tenth of a second is slightly slower than some special hardware might need, but since EXACTF eliminates almost all the light-pen circuitry and since it is faster than most people can react, it is most useful.

In Enhancement No. 5, we will look at mixed fields, a brand new and most exciting use of EXACTF. With mixed fields, you can mix and match text, HIRES, and LORES in any combination anywhere on the screen, as well as doing screen splits and dynamic video wipes.

You can also "inside out" your EXACTF to find the exact start of blanking if you want to. We have shown this as module 6 and named it EXACTB. This time, you exit exactly 7 CPU clocks into the vertical blanking rather than 7 CPU clocks into the next field.

One limit to EXACTF or EXACTB is that you cannot use either one each and every field, like you could the earlier subroutines. The reason for this is that up to seven fields may be needed in order to back up to the exact timing exit point we are after. This means that you have to keep your exact lock till you are finished with it. During the time you keep the lock, each and every CPU clock cycle must be exactly accounted for.

For instance, for light-pen use, after you acquire an exact lock, you have to increment software counters for line and position measurement and do whatever else you need to do to end up with an exact measurement. If you are doing a horizontal video split, or other mixed field, your timing must remain controlled and constant till you are done. This means that you probably will want to either time out your split or else test for a pressed key to exit. Your CPU will be tied up till you are done with the display. We will see just how to handle this timing situation and how to make room for other program time problems in the next enhancement.

The point is that once you do an exact lock with EXACTF or EXACTB, your CPU is 100% committed to the reason why you wanted to lock until you are finished. This is unlike the earlier subroutines that quickly return control back to your main programs.

We will note in passing that you can do an exact lock in a fraction of a field, rather than in seven fields, if you are willing to add extra hardware. This greatly simplifies program coding. Details on this appear in Enhancement 13 of Volume 2.

Let's sum all this up.

THE FIELD SYNC MODULES

CRUDE —Exits sometime during the vertical blanking time. It probably will bomb if the FS mod is absent. It may be used each field but must not be called twice in a field.

FEDGE —Exits on the start of a new field with some jitter. It will bomb if the FS mod is absent. It may be used each field.

BEDGE —Like *FEDGE*, only exits on start of blanking.

ALTFLD —Switches between text and graphics fields continuously until any key is pressed. Needs *FEDGE* to run, and will bomb if the FS mod is absent.

EXACTF —Finds the **exact** start of a field with zero jitter. May take up to seven fields to do a perfect lock. Will bomb if FS mod is absent. Lock timing must be continued by calling program.

EXACTB —Like *EXACTF*, only exits on exact blanking start.

To rehash, use *CRUDE* for simple field switching if you are sure you won't be calling it more than once per field. Use *FEDGE* when you want the start of a field with no danger of catching the same field twice. Use *BEDGE* if you need a medium amount of animation per field.

Use the *ALTFLD* for simple text and graphics overlays. Use *EXACTF* when you want to find the start of a field exactly for use with a light pen, touch screen, horizontal video split, or horizontal video wipe. Or, use *EXACTB* to find an exact screen location and, then, do lots of setup before the live scan time actually starts.

We will see more examples of how to use your field sync in upcoming enhancements. Note that we have kept each sync module separate so you can add to them or move them anyway you like. The funny misspellings are done so that each module has its own unique label inside a common assembly-language source program.

While we have shown an RTS, or Return from Subroutine, at the end of each module, you are free to move any RTS down and stuff as much other machine-language code in the subroutine as you like.

Program number 4-2 shows us a quick way to test your field sync modification. This Applesoft program loads *ALTFLD* and *FEDGE*. It then prints a message on Text I and some artwork on HIRES 1. *ALTFLD* is then called on to superimpose text and HIRES. The text-over-HIRES display continues till you hit any key.

Your field sync is working if you get the double text and HIRES image with no glitches or noise anywhere on the screen. Some flicker is normal with this

PROGRAM 4-2
FIELD SYNC QUICK TEST

LANGUAGE: APPLESOFT

NEEDS: FIELD SYNC MOD
FEDGE SUB
ALTFLD SUB

```
10 REM *****
12 REM *
14 REM *   FIELD SYNC   *
16 REM *   QUICK TEST   *
18 REM *
20 REM *   VERSION 1.0   *
22 REM *
24 REM *   COPYRIGHT 1981 *
26 REM * BY DON LANCASTER *
28 REM * AND SYNERGETICS *
30 REM *
32 REM * ALL COMMERCIAL *
34 REM * RIGHTS RESERVED *
36 REM *
38 REM *****

52 REM FIELD SYNC DESCRIBED
54 REM IN ENHANCING YOUR
56 REM APPLE II, VOLUME I

80 REM NEEDS ALTFLD AND
82 REM FEDGE FIELD SYNC
84 REM SUBROUTINES.

100 PRINT "BLOAD FIELD SYNC UTIL
    ITY SUBS": REM CTRL D

200 HOME : VTAB 4: REM TEXT
210 PRINT "YOUR FIELD SYNC": PRINT
    "IS WORKING IF ..."
220 VTAB 10: HTAB 14: PRINT "THE
    SE WORDS": HTAB 14: PRINT "
    ARE IN A BOX"
230 VTAB 16: HTAB 20: PRINT "...
    AND THERE ARE NO
240 HTAB 20: PRINT "    GLITCHES
    ANYWHERE";: HTAB 20: PRINT
    "    ON THE SCREEN.": REM
```

PROGRAM 4-2, CONT'D...

```
250 HGR : HCOLOR= 5: POKE - 163
    02,0: REM GRAPHICS
260 HPLOT 80,64 TO 184,64 TO 184
    ,98 TO 80,98 TO 80,64
270 HPLOT 81,65 TO 185,65 TO 185
    ,99 TO 81,99 TO 81,65
280 HCOLOR= 1: HPLOT 31,12 TO 31
    ,81 TO 75,81 TO 71,76 TO 75,
    81 TO 71,86 TO 71,76
290 HPLOT 29,11 TO 29,16 TO 33,1
    1 TO 33,16: HPLOT 73,78: HPLOT
    73,82: HPLOT 73,80
300 HCOLOR= 2: HPLOT 191,81 TO 2
    36,81 TO 236,116 TO 232,112 TO
    240,112 TO 236,116
310 HPLOT 189,80 TO 197,80 TO 18
    9,82 TO 197,82: HPLOT 236,11
    3: REM

400 FOR N = 1 TO 3000: NEXT N: PRINT
    "": REM BELL CTRL G
410 TEXT : FOR N = 1 TO 3000: NEXT
    N: PRINT "": REM BELL CT
    RL G

500 CALL 816: REM CALL ALTFLD

990 POKE - 16368,0: REM RESET
    KEY STROBE
995 PRINT : PRINT "RUN MENU": REM
    EXIT ON KP
996 REM DELETE 995 IF AUTO MENU
    IS NOT IN USE

999 END
```

particular use of field sync. You can minimize this flicker by using black and white rather than color displays, or by using darker colors rather than white, or by minimizing large blocks of text, or by keeping the total screen information and display time at a minimum. Proper setting of contrast and brightness on the display will also make a big difference.

Once again, this apparent flicker applies only to some field alternators. Practically all other uses of field sync are glitch and flicker free.

Note that your field sync must have exclusive use of the cassette read circuitry and SW3 when it is in use. We have used the phantom "SW3" input of the cassette read circuitry instead of the more obvious game paddle SW2 because many word processor programs will use SW2 to route a SHIFT key command into their programs.

You have to break your field sync connection any time you use your cassette input. Direct entry of the blanking waveform into the IN jack of the cassette is not recommended—first, because capacitor C10 is too small and, more crucially, because the long and erratic delay in analog amplifier K13 would give unpredictable results in precision field sync applications which need an exact lock.

Precision field sync is so important and so exciting that it should be immediately added to your Apple and done in a simple, standard, and easy-to-use way. Let us know what new uses you find for this exciting new capability 🍏

The Apple Assembler source and object programs FIELD SYNC UTILITY SUBS.SOURCE and FIELD SYNC UTILITY SUBS, along with the Applesoft program FIELD SYNC TESTER are included on the companion diskette to this volume. All three programs are fully copyable.

A complete set of all parts needed to build one field sync modification is included in the companion parts kit to this volume.

Enhancement



FUN WITH MIXED FIELDS

Mix TEXT, HIRES, AND LORES anywhere on the screen using your field sync mod and some simple support software. Displays are glitch and flicker free and open up many exciting new uses. The secret of the 121 LORES colors is also revealed. Or, is it?

FUN WITH MIXED FIELDS

How would you like to be able to mix and match text, LORES, and HIRES together *anywhere* you like on your Apple screen? It's a lot easier than you might think. All it takes is some simple support software that we'll look at here. We'll use it along with the one-wire field sync mod of Enhancement 4.

Now you can freely mix text and LORES color, even off of the same display page, and anywhere you like on the screen. You can title HIRES displays at the top and sides, mix HIRES and LORES together, and label LORES bar graphs. You can inset and quickly change normal, inverse, or flashing text anywhere on a HIRES or LORES display.

Think of the possibilities! And, the mix-and-match magic of *mixed fields* can be done much simpler and much faster than can be accomplished by using a

straight HIRES display. Your memory needs also can be much smaller with mixed fields. And, the basic idea behind mixed fields can be easily extended into mind-boggling things like video wipes, whole new worlds of animation, external video special effects, and true 3-D vibrating mirror displays.

What is a mixed field?

MIXED FIELD —An Apple video display mode that changes between text, LORES, and HIRES on the fly, letting you mix and match what goes on the screen at any instant.

The exciting thing about mixed fields is that everything is there at once. Since the screen is switched “on the fly,” there is no flicker like you sometimes get with alternating field displays.

As with any Apple feature, mixed fields have some limits and disadvantages. The first is that you have to very carefully set up a *display file* that is part of a display program. This display file controls when modes get changed on the screen. The display file isn’t nearly as bad as a shape table, but you still must be able to use and understand one before you can get mixed fields to work for you.

The second limiting factor to mixed field displays is that they take up the lion’s share of the CPU time when you are using them. Mixed field displays are better suited for titles, transitions, and displaying results than for use where you must do a lot of real-time computing. We have built in an automatic display timer and an option to exit on any key pressed. These give you the continuity you will need to use mixed fields. You can easily switch between mixed fields and regular displays at any time in a program.

You also have the option of using up to 4300 or so clock cycles per field for things like limited animation, character entry, and so on. The available throughput ends up around one quarter of normal with this option. Use of this nondisplay time is tricky, since you must either maintain an exact number of clock cycles or else resynchronize each field.

The third limit to mixed fields is that you can get some very awful glitches on the screen if you aren’t careful. The absolute worst source of the glitches is caused by a change Apple made back in Revision 1. This one is easily eliminated with the \$3.00 *Glitch Stomper* mod upcoming in Enhancement 6. The remaining glitches are easily made invisible once you understand what causes them. You can even make the glitches work for you once you really get to know them.

All in all, though, these limits to mixed fields are easy to get around if you spend the time and trouble to understand what field switching is all about. And, the spectacular results certainly make it all worthwhile.

How mixed fields work

The following example will show how mixed field displays work. The general idea behind mixed fields is to flip soft switches in *exactly* the same screen position for every successive field. In this example, we do a graph, starting with a one-line text title at the top. Then, we do a single LORES horizontal grey line separating title from graph. The graph itself consists of Y-axis text on the left and

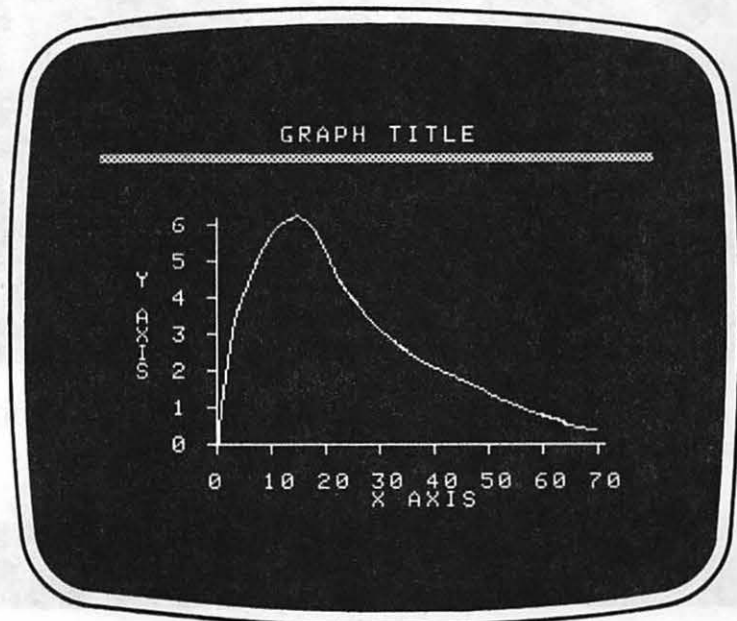
HIRES lines and points on the right. Finally, we do the X-axis stuff in straight text. For each and every field, soft switches are flipped in just the right locations to make all this happen.

In order to understand how to do a mixed field display, it is necessary that you fully understand the following procedure.

A mixed field display needs the field sync mod of the previous enhancement and an exact locking program called VFFS. Inside VFFS are three sets of files, called CONTROL, HPAT, and VPAT. These files are changed to suit your needs.

For instance, suppose you want to do a fancy graph with a title at the top, then a grey bar, then a HIRES graph with ordinary text as the Y axis, then a two-line text X axis below.

Something like this . . .

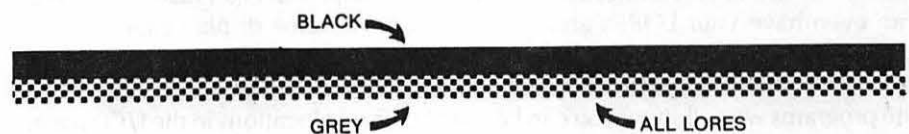


Split the display up into horizontal pieces, starting from the top. Each piece has to have an identical horizontal pattern on each and every one of its scan lines. With VFFS, you are allowed to have four different horizontal patterns. These four horizontal patterns can repeat in order as often as you like.

Our first horizontal pattern is used for the title. It simply switches to text and keeps us in text for the eight lines that are needed to put down a row of characters

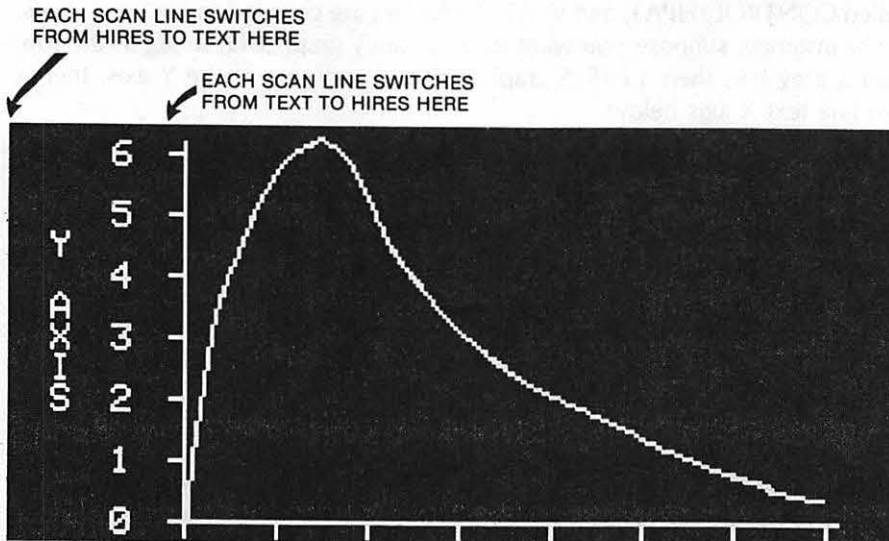


Our second horizontal pattern is used for the grey bar. It simply switches us to LORES and keeps us in LORES for another 8 lines. We'll use 4 lines for some black space under the title, and the second 4 lines for the grey bar.

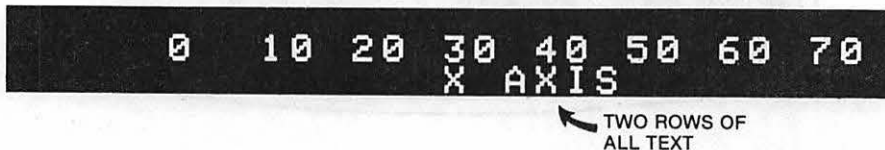


Your text and LORES can be off the same display page as long as you use multiples of 8 scan lines per mode.

Our third horizontal pattern is more complicated. It switches to text at the start of each horizontal line and then switches to HIRES on the eighth character of each and every horizontal scan line. This continues for most of the display, down to line number 175.



Our final horizontal pattern goes back to all text, and lasts for the final 16 scan lines, going from line 175 to the bottom of the screen at line 191.



VFFS first does an exact lock and, then, repeats its formatted display every field. The display ends on a pressed key, a selected timeout, or both.

This particular example is comparable to VFFS.GRAPH that is used in demonstration Program 5-2. Each new set of file values should be saved under its own unique name.

In the preceding example, any time you read or write to memory location \$C051, the display will immediately switch to its text mode. Whenever you read or write to memory location \$C050, the display immediately switches to its graphics mode. If you flipped the \$C051 switch at the top of the screen and then flipped \$C050 after 8 scan lines went by, you would end up with a line of text at the top of the screen and graphics the rest of the way down.

Why the eighth line? Because there are eight horizontal raster lines needed per character, since each character is made up vertically of seven live dots and a blank undot. The number of raster lines usually needed equals eight times the number of text lines. As long as you stick with multiples of eight raster lines, you can even have your LORES and text come off the same display page!

Other switches will let you flip between HIRES and LORES, between memory page 1 and memory page 2, and between full and mixed graphics. And, since the programs we will show you can hit many different locations in the I/O space,

you can also flip the annunciator soft switches or read push-button inputs. This opens the door to special video effects, singling out one display line for analysis, doing animation and wipes, using precision hardware-free light pens, doing touch screens, adding grey scale and anti-aliasing, providing sync for 3-D vibrating mirror displays, and so on.

Some very simple field mixing can be done by flipping a soft switch or two at any old time during the vertical blanking interval. For fancier field mixes, we have to lock to the field start *exactly*, using EXACTF or something similar. An exact lock is needed anytime you want to *change* something in the *middle* of a horizontal scan line.

To use field sync, you set up a switch-flipping program that takes exactly one field to complete. You then find the start of a field with EXACTF and, then, drop into this switch-flipping program. The program then keeps changing the display back and forth for you, putting what you want exactly where you want it. This will continue until either an automatic timeout is completed or until a key is optionally pressed.

You can write lots of very simple mixed field programs. One could give you changes at the beginning of any horizontal line. Another could give you changes along any horizontal line. Yet another could give you a way to inset text, and so on.

Instead of all these relatively simple and specialized programs, we will show you one fairly fancy machine-language subroutine called the *Video Field Formatter Sub*, or VFFS for short. We will start out with a do-nothing VFFS called VFFS.EMPTY. You will then customize the files in this VFFS to do whatever you like and, then, rename it. Examples that we will see will include VFFS.BOXES, VFFS.GRAPH, and so on.

VFFS.EMPTY appears as Program 5-1 and can be used to flip up to 11 switches on each and every one of 192 live scan lines for a total of up to 2112 display changes per field.

Probably the best way to understand VFFS is to jump in with both feet and use it. An Applesoft demo, called Fun With Field Switches, is shown as Program 5-2. This demo shows you some mixed field animation, a HIRES plot with text vertical titling, a LORES bar graph with text callouts, and a final display that mixes three alphabets for you.

VFFS.EMPTY is a machine-language program that needs a total of around 520 bytes, including its working files. We have stashed it starting at \$8AFF, or decimal 35583. You must protect this space from either Applesoft or Integer BASIC use. The HIMEM command can do this for you, either early in an Applesoft program or before loading an Integer program. I used this space because you can optionally load and protect VFFS in the highest optional character slot under Apple's HRCG Hi-Res Character Generator.

You can easily relocate VFFS anywhere you like, but any relocation will change all the file locations and design worksheets we are about to look at.

Working files

The keys to VFFS lie in three sets of *files*. The simplest file is a one-byte file called CONTROL. CONTROL sets the length of time that the display is active and decides whether a keypressed exit is allowed. CONTROL is located at \$8CFB (decimal 36091). Figure 5-1 shows us how CONTROL is used.

The most significant bit of CONTROL is *set* to a *one* to *activate* the keypressed exit, and *cleared* to a *zero* to *disable* the keypressed exit. The next most significant bit of CONTROL is *set* to a *one* to *activate* the automatic timeout or *cleared* to *zero* for an "infinite" display. The remaining six lowest

PROGRAM 5-1

VIDEO FIELD FORMATTER SUB. VFFS. EMPTY

LANGUAGE: APPLE ASSEMBLY

NEEDS: FIELD SYNC MOD
HIMEN < 35583

```

8AFF:      4 ; *****
8AFF:      5 ; *
8AFF:      6 ; * VIDEO FIELD *
8AFF:      7 ; * FORMATTER SUB *
8AFF:      8 ; * VFFS.EMPTY *
8AFF:      9 ; *
8AFF:     10 ; * ($8AFF.8D07) *
8AFF:     11 ; *
8AFF:     12 ; * VERSION 1.0 *
8AFF:     13 ; * ( 8-25-81) *
8AFF:     14 ; *
8AFF:     15 ; * COPYRIGHT 1981 *
8AFF:     16 ; * BY DON LANCASTER *
8AFF:     17 ; * AND SYNERGETICS *
8AFF:     18 ; *
8AFF:     19 ; * ALL COMMERCIAL *
8AFF:     20 ; * RIGHTS RESERVED *
8AFF:     21 ; *
8AFF:     22 ; *****

8AFF:     24 ; THIS PROGRAM LETS YOU MIX
8AFF:     25 ; AND MATCH SCREEN MODES IN
8AFF:     26 ; MANY DIFFERENT COMBINATIONS.

8AFF:     28 ; SCREEN MODE CAN BE CHANGED
8AFF:     29 ; ONCE EACH HBLANK TIME USING
8AFF:     30 ; FILE "VPATRN".

8AFF:     32 ; SCREEN MODE CAN BE CHANGED
8AFF:     33 ; UP TO TEN TIMES PER H SCAN
8AFF:     34 ; USING FILES "HPAT1" THROUGH
8AFF:     35 ; "HPAT4". FOUR OR FEWER
8AFF:     36 ; H PATTERNS ARE ALLOWED.

8AFF:     38 ; SEE ENHANCEMENT #5 OF
8AFF:     39 ; ENHANCING YOUR APPLE II,
8AFF:     40 ; VOL I FOR FULL USE DETAILS.

8AFF:     42 ; TO USE, SET UP PATTERN FILES
8AFF:     43 ; THEN JSR $8B00 FROM MACHINE
8AFF:     44 ; LANGUAGE OR CALL 35584 FROM
8AFF:     45 ; APPLESOFT.

```

PROGRAM 5-1, CONT'D...

C060:	48 DUMMY	EQU	\$C060	; LOCATION FOR NO-SWITCH
C000:	49 KEYBD	EQU	\$C000	; KEY PRESS CHECK
C060:	50 SYNC	EQU	\$C060	; VBLANK VIA "SW3"
C000:	51 SWITCH	EQU	\$C000	; VFILE SWITCH LOCATION
FCA8:	52 WAIT	EQU	\$FCA8	; MONITOR DELAY SUB

PROGRAM 5-1, CONT'D...

```

8AFF:          55 ;   **** EXACT LOCK TO FIELD START ****

8AFF:EA        57      NOP          ; EVEN PAGE START (FOR HRCG)

8B00:20 FE 8C   59 START  JSR  SETUP   ; INITIALIZE KEYBOARD AND
                                     TIMEOUT

8B03:2C 60 C0   61 EXACTF  BIT  SYNC    ; LOCK TO FIELD EDGE
8B06:10 FB      62      BPL  EXACTF    ;
8B08:2C 60 C0   63 BLENK   BIT  SYNC    ; FIND FIELD START
8B0B:30 FB      64      BMI  BLENK     ; WITH JITTER
8B0D:EA        65      NOP          ; STALL FOR 2 CYCLES
8B0E:10 00      66      BPL  STALL     ; STALL FOR 3 CYCLES
8B10:A9 3B      67 STALL  LDA  #$3B    ; DELAY FOR 17029
8B12:20 A8 FC   68      JSR  WAIT     ; CLOCK CYCLES TOTAL
8B15:A9 34      69      LDA  #$34    ; USING THE MONITOR
8B17:20 A8 FC   70      JSR  WAIT     ; DELAY ROUTINE AND
8B1A:A9 01      71      LDA  #$01    ; THIS LOOP TIMING.
8B1C:20 A8 FC   72      JSR  WAIT     ;
8B1F:2C 60 C0   73      BIT  SYNC    ; HAVE WE BACKED TO START?
8B22:10 EC      74      BPL  STALL     ; NO, GO BACK ONE MORE

8B24:          76 ;   **** START OF FIELD ****

8B24:A0 C0      78 NEWFLD LDY  #$C0    ; FOR 192 LINES

8B26:B9 00 8C   80 NXTLN1 LDA  VPATRN,Y ; GET LINE PATTERN
8B29:30 63      81      BMI  HPAT2
8B2B:10 00      82 HPAT1  BPL  HP1
8B2D:29 7F      83 HP1   AND  #$7F    ; MASK SWITCH COMMAND
8B2F:AA        84      TAX
8B30:9D 00 C0   85      STA  SWITCH,X ;
8B33:8D 60 C0   86      STA  DUMMY    ;
8B36:8D 60 C0   87      STA  DUMMY    ;
8B39:8D 60 C0   88      STA  DUMMY    ; *
8B3C:8D 60 C0   89      STA  DUMMY    ; **
8B3F:8D 60 C0   90      STA  DUMMY    ; *
8B42:8D 60 C0   91      STA  DUMMY    ; *
8B45:8D 60 C0   92      STA  DUMMY    ; *
8B48:8D 60 C0   93      STA  DUMMY    ; *
8B4B:8D 60 C0   94      STA  DUMMY    ; ***
8B4E:8D 60 C0   95      STA  DUMMY    ;
8B51:88        96      DEY          ; ONE LESS LINE
8B52:F0 32      97      BEQ  BOTTOM    ; AT SCREEN BOTTOM?
8B54:D0 D0      98      BNE  NXTLN1

```

PROGRAM 5-1, CONT'D...

```

8B56:B9 00 8C 102 NXTLN4 LDA VPATRN,Y ; GET LINE PATTERN
8B59:30 D0      103      BMI HPAT1    ;
8B5B:10 00      104 HPAT4 BPL HP4     ;
8B5D:29 7F      105 HP4  AND #$7F     ; MASK SWITCH COMMAND
8B5F:AA          106      TAX          ;
8B60:9D 00 C0   107      STA SWITCH,X ;
8B63:8D 60 C0   108      STA DUMMY    ;
8B66:8D 60 C0   109      STA DUMMY    ;
8B69:8D 60 C0   110      STA DUMMY    ;      *
8B6C:8D 60 C0   111      STA DUMMY    ;      **
8B6F:8D 60 C0   112      STA DUMMY    ;      * *
8B72:8D 60 C0   113      STA DUMMY    ;      * *
8B75:8D 60 C0   114      STA DUMMY    ;      *****
8B78:8D 60 C0   115      STA DUMMY    ;      *
8B7B:8D 60 C0   116      STA DUMMY    ;      *
8B7E:8D 60 C0   117      STA DUMMY    ;
8B81:88          118      DEY          ; ONE LESS LINE
8B82:F0 02      119      BEQ BOTTOM    ; AT SCREEN BOTTOM?
8B84:D0 D0      120      BNE NXTLN4

8B86:4C C2 8C 122 BOTTOM JMP BOTTM1   ; "SPlice" RELATIVE BRANCH

8B89:B9 00 8C 124 NXTLN2 LDA VPATRN,Y ; GET LINE PATTERN
8B8C:30 30      125      BMI HPAT3    ;
8B8E:10 00      126 HPAT2 BPL HP2     ;
8B90:29 7F      127 HP2  AND #$7F     ; MASK SWITCH COMMAND
8B92:AA          128      TAX          ;
8B93:9D 00 C0   129      STA SWITCH,X ;
8B96:8D 60 C0   130      STA DUMMY    ;
8B99:8D 60 C0   131      STA DUMMY    ;
8B9C:8D 60 C0   132      STA DUMMY    ;
8B9F:8D 60 C0   133      STA DUMMY    ;      ***
8BA2:8D 60 C0   134      STA DUMMY    ;      *   *
8BA5:8D 60 C0   135      STA DUMMY    ;      *
8BA8:8D 60 C0   136      STA DUMMY    ;      *
8BAB:8D 60 C0   137      STA DUMMY    ;      *
8BAE:8D 60 C0   138      STA DUMMY    ;      *
8BB1:8D 60 C0   139      STA DUMMY    ;      *****
8BB4:88          140      DEY          ; ONE LESS LINE
8BB5:F0 CF      141      BEQ BOTTOM    ; AT SCREEN BOTTOM?
8BB7:D0 D0      142      BNE NXTLN2   ;

```

PROGRAM 5-1, CONT'D...

```

8BB9:B9 00 8C 146 NXTLN3 LDA VPATRN,Y ; GET LINE PATTERN
8BBC:30 9D 147 BMI HPAT4
8BBE:10 00 148 HPAT3 BPL HP3 ;
8BC0:29 7F 149 HP3 AND #$7F ; MASK SWITCH COMMAND
8BC2:AA 150 TAX ;
8BC3:9D 00 C0 151 STA SWITCH,X ;
8BC6:8D 60 C0 152 STA DUMMY ;
8BC9:8D 60 C0 153 STA DUMMY ;
8BCC:8D 60 C0 154 STA DUMMY ; *****
8BCF:8D 60 C0 155 STA DUMMY ; *
8BD2:8D 60 C0 156 STA DUMMY ; *
8BD5:8D 60 C0 157 STA DUMMY ; **
8BD8:8D 60 C0 158 STA DUMMY ; *
8BDB:8D 60 C0 159 STA DUMMY ; * *
8BDE:8D 60 C0 160 STA DUMMY ; ***
8BE1:8D 60 C0 161 STA DUMMY ;
8BE4:88 162 DEY ; ONE LESS LINE
8BE5:F0 9F 163 BEQ BOTTOM ; AT SCREEN BOTTOM?
8BE7:D0 D0 164 BNE NXTLN3 ;

```

```

8BE9: 167 ; ***** VBLANKING DELAY *****

```

```

8BE9:A9 20 169 VBSTAL LDA #$20 ; DELAY FOR 4472
8BEB:20 A8 FC 170 JSR WAIT ; CPU CYCLES
8BEE:A9 14 171 LDA #$14 ;
8BF0:20 A8 FC 172 JSR WAIT ;
8BF3:A9 06 173 LDA #$06 ;
8BF5:20 A8 FC 174 JSR WAIT ;
8BF8:60 175 RTS ;

```

PROGRAM 5-1, CONT'D...

8C00:60 60 60	179	VPATRN	DFB	96,96,96,96,96,96,96,96, 96,96,96,96,96,96,96,96
8C03:60 60 60				
8C06:60 60 60				
8C09:60 60 60				
8C0C:60 60 60				
8C0F:60				
8C10:60 60 60	180		DFB	96,96,96,96,96,96,96,96, 96,96,96,96,96,96,96,96
8C13:60 60 60				
8C16:60 60 60				
8C19:60 60 60				
8C1C:60 60 60				
8C1F:60				
8C20:60 60 60	181		DFB	96,96,96,96,96,96,96,96, 96,96,96,96,96,96,96,96
8C23:60 60 60				
8C26:60 60 60				
8C29:60 60 60				
8C2C:60 60 60				
8C2F:60				
8C30:60 60 60	182		DFB	96,96,96,96,96,96,96,96, 96,96,96,96,96,96,96,96
8C33:60 60 60				
8C36:60 60 60				
8C39:60 60 60				
8C3C:60 60 60				
8C3F:60				
8C40:60 60 60	183		DFB	96,96,96,96,96,96,96,96, 96,96,96,96,96,96,96,96
8C43:60 60 60				
8C46:60 60 60				
8C49:60 60 60				
8C4C:60 60 60				
8C4F:60				
8C50:60 60 60	184		DFB	96,96,96,96,96,96,96,96, 96,96,96,96,96,96,96,96
8C53:60 60 60				
8C56:60 60 60				
8C59:60 60 60				
8C5C:60 60 60				
8C5F:60				
8C60:60 60 60	185		DFB	96,96,96,96,96,96,96,96, 96,96,96,96,96,96,96,96
8C63:60 60 60				
8C66:60 60 60				
8C69:60 60 60				
8C6C:60 60 60				
8C6F:60				
8C70:60 60 60	186		DFB	96,96,96,96,96,96,96,96, 96,96,96,96,96,96,96,96
8C73:60 60 60				
8C76:60 60 60				
8C79:60 60 60				
8C7C:60 60 60				
8C7F:60				

PROGRAM 5-1, CONT'D...

```

8C80:60 60 60 187      DFB  96,96,96,96,96,96,96,96,
                        96,96,96,96,96,96,96,96
8C83:60 60 60
8C86:60 60 60
8C89:60 60 60
8C8C:60 60 60
8C8F:60
8C90:60 60 60 188      DFB  96,96,96,96,96,96,96,96,
                        96,96,96,96,96,96,96,96
8C93:60 60 60
8C96:60 60 60
8C99:60 60 60
8C9C:60 60 60
8C9F:60
8CA0:60 60 60 189      DFB  96,96,96,96,96,96,96,96,
                        96,96,96,96,96,96,96,96
8CA3:60 60 60
8CA6:60 60 60
8CA9:60 60 60
8CAC:60 60 60
8CAF:60
8CB0:60 60 60 190      DFB  96,96,96,96,96,96,96,96,
                        96,96,96,96,96,96,96,96
8CB3:60 60 60
8CB6:60 60 60
8CB9:60 60 60
8CBC:60 60 60
8CBF:60
8CC0:60 60      191      DFB  96,96

8CC2:      193 ;      **** KEYPRESSED AND TIMEOUT ****

8CC2:20 CB 8C 195 BOTTM1 JSR  KEYTIME      ; TAKE CARE OF EXIT
8CC5:20 E9 8B 196      JSR  VBSTAL      ; DELAY TILL NEXT FIELD
8CC8:4C 24 8B 197      JMP  NEWFLD

```

PROGRAM 5-1, CONT'D...

```

8CCB:2C FB 8C 201 KEYTIME BIT CONTROL ; IS KEY EXIT ACTIVE?
8CCE:10 08 202 BPL NOKEY ;
8CD0:2C 00 C0 203 BIT KEYBD ; LOOK FOR KEY
8CD3:10 06 204 BPL KEYOK ; NOT THERE?
8CD5:68 205 PLA ; POP SUBROUTINE
8CD6:68 206 PLA ;
8CD7:60 207 RTS ; EXIT
8CD8:EA 208 NOKEY NOP ; EQUALIZE 6
8CD9:EA 209 NOP ;
8CDA:EA 210 NOP ;
8CDB:EE FD 8C 211 KEYOK INC TIMEX ; INCREMENT TIMEOUT
MULTIPLIER
8CDE:2C FB 8C 212 BIT CONTROL ; IS TIMER ACTIVE?
8CE1:50 0F 213 BVC NOTIME ;
8CE3:A9 1F 214 LDA #$1F ; MASK FOR 1/64
8CE5:2D FD 8C 215 AND TIMEX ; AND TEST MULTIPLIER
8CE8:D0 0C 216 BNE NOMULT
8CEA:CE FC 8C 217 DEC TIMER ; ONE LESS COUNT
8CED:D0 0B 218 BNE TIMEOK ; DONE?
8CEF:68 219 PLA ; POP SUBROUTINE
8CF0:68 220 PLA ;
8CF1:60 221 RTS ; EXIT
8CF2:EA 222 NOTIME NOP ; EQUALIZE 8
8CF3:EA 223 NOP ;
8CF4:EA 224 NOP ;
8CF5:EA 225 NOP ;
8CF6:EA 226 NOMULT NOP ; EQUALIZE 8
8CF7:EA 227 NOP ;
8CF8:EA 228 NOP ;
8CF9:EA 229 NOP ;
8CFA:60 230 TIMEOK RTS ; RETURN TO NEXT SCAN

8CFB:C4 232 CONTROL DFB $C4 ; ARMS KP AND SETS TIMEOUT
8CFC:00 233 TIMER DFB $00 ; COUNTER FOR TIMEOUT
8CFD:00 234 TIMEX DFB $00 ; TIMEOUT * 64 MULTIPLIER

8CFE:AD FB 8C 236 SETUP LDA CONTROL ; INITIALIZE TIMEOUT
8D01:29 3F 237 AND #$3F ; MASK TIMEOUT BITS
8D03:8D FC 8C 238 STA TIMER
8D06:60 239 RTS ; AND CONTINUE

```

*** SUCCESSFUL ASSEMBLY: NO ERRORS

PROGRAM 5-2 FUN WITH MIXED FIELDS

LANGUAGE: APPLESOFT

NEEDS: FIELD SYNC MOD
HIMEM < 35583
VFFS. BOXES
VFFS. GRAPH
VFFS. GIRLS
VFFS. BYE

```

10 REM *****
12 REM *
14 REM *      FUN WITH      *
16 REM *    MIXED FIELDS    *
18 REM *
20 REM *    VERSION 1.0      *
22 REM *    (9-26-81)       *
23 REM *
24 REM *    COPYRIGHT 1981   *
26 REM * BY DON LANCASTER  *
28 REM * AND SYNERGETICS    *
30 REM *
32 REM * ALL COMMERCIAL     *
34 REM * RIGHTS RESERVED    *
36 REM *
38 REM *****

50 REM THIS PROGRAM SHOWS YOU
52 REM HOW TO MIX AND MATCH
54 REM TEXT, LORES, AND HIRES
56 REM ANYWHERE ON THE SCREEN
58 REM IN ANY COMBINATION.

60 REM THE FIELD SYNC HARDWARE
62 REM MOD AND SUBROUTINES
64 REM VFFS.BOXES, VFFS.GRAPH,
66 REM VFFS.GIRLS AND VFFS.BYE
68 REM ARE NEEDED.

70 REM SEE ENHANCEMENTS #4,
72 REM #5, AND #6 OF ENHANCING
74 REM YOUR APPLE II, VOL I
76 REM FOR MORE USE DETAILS.

100 HIMEM: 35500: REM  PROTECT
    VFFS SPACE
110 HGR : TEXT : HOME : GR : REM
    INITIALIZE ALL

```

PROGRAM 5-2, CONT'D...

```

1000 REM ** TITLE BOXES **

1004 PRINT
1005 PRINT "BLOAD VFFS.BOXES": REM
    CTRL D
1010 SPEED= 10
1020 VTAB 8: HTAB 9: PRINT " FU
    N ";: HTAB 25: PRINT " WI
    TH ";
1030 VTAB 15: HTAB 9: PRINT " MI
    XED ";: HTAB 25: PRINT " FI
    ELDS "
1040 SPEED= 255: REM

1050 COLOR= 1
1080 FOR N = 1 TO 3500: NEXT N
1090 HLIN 6,16 AT 11: HLIN 22,32
    AT 11: HLIN 6,16 AT 18: HLIN
    22,32 AT 18
1100 HLIN 6,16 AT 25: HLIN 22,32
    AT 25: HLIN 6,16 AT 32: HLIN
    22,32 AT 32
1110 VLIN 11,18 AT 6: VLIN 25,32
    AT 6: VLIN 11,18 AT 16: VLIN
    25,32 AT 16
1120 VLIN 11,18 AT 22: VLIN 25,3
    2 AT 22: VLIN 11,18 AT 32: VLIN
    25,32 AT 32
1125 REM

1130 PRINT "": FOR N = 1 TO 3500
    : NEXT N
1140 PRINT "": POKE 36091,212: CALL
    35584
1145 REM

1150 HGR : HCOLOR= 2: POKE 49234
    ,0
1160 HPLOT 12,8 TO 258,8 TO 259,
    85 TO 12,85 TO 12,8
1169 POKE 49168,0
1170 HPLOT 12,90 TO 258,90 TO 25
    8,170 TO 12,170 TO 12,90
1180 FOR N = 1 TO 3500: NEXT N: PRINT
    ""
1190 CALL 35584: REM MIX FIELDS

```

PROGRAM 5-2, CONT'D...

```

2000 REM  ** GRAPH AND TITLE **

2006 H = 62:V = 160
2010 TEXT : HOME : SPEED= 100
2015 PRINT "BLOAD VFFS.GRAPH": REM
      CTRL D
2020 PRINT "          DIPTHONG-SNORG
      EL CORRELATIONS:"
2030 PRINT " .....
      ..... "
2040 PRINT "      40 -"
2050 PRINT : PRINT "      -"
2060 PRINT : PRINT " S  30 -"
2070 PRINT " N      "
2080 PRINT " O      -"
2090 PRINT " R      "
2100 PRINT " G  20 -"
2110 PRINT " E      "
2120 PRINT " L      -"
2130 PRINT " S      "
2140 PRINT "      10 -"
2150 : PRINT : PRINT "      -"
2160 PRINT : PRINT "      0 -"
2170 PRINT : PRINT "      0
1    2    3    4    5    6    7"
2180 PRINT : PRINT "
      DIPTHONGS";
2190 VTAB 1: HTAB 0: PRINT " ": SPEED= 255
2191 HGR : HCOLOR= 1: POKE 49234
      ,0
2192 HPLOT 63,156 TO 255,156
2193 HCOLOR= 1: HPLOT 87,157 TO
      87,165: HPLOT 115,157 TO 115
      ,165: HPLOT 143,157 TO 143,1
      65
2194 HPLOT 171,157 TO 171,165: HPLOT
      199,157 TO 199,165: HPLOT 22
      7,157 TO 227,165: HPLOT 255,
      157 TO 255,165
2210 GOSUB 20000: REM PLOT
2220 HCOLOR= 7: HPLOT 229,32 TO
      229,120
2300 POKE 36091,208: CALL 35584
2400 FLASH : VTAB 5: HTAB 34: PRINT
      "1066"
2410 VTAB 15: HTAB 34: PRINT "14
      92"
2418 HCOLOR= 7: HPLOT 229,28 TO
      229,128
2420 NORMAL : CALL 35584: REM M
      IX FIELDS
2999 REM

```

PROGRAM 5-2, CONT'D...

```
3000 REM ** GIRLS **

3010 GR : CALL - 1998: POKE -
      16302,0
3020 PRINT "BLOAD VFFS.GIRLS": REM

3100 FOR N = 1 TO 10: READ L: COLOR=
      N: HLIN 0,L AT (4 * N - 1): FOR
      K = 1 TO 200: NEXT K,N
3105 : FOR N = 1 TO 2000: NEXT N
3107 HGR : HCOLOR= 3: HPLOT 4,17
      4 TO 240,174 TO 243,179 TO 2
      46,169 TO 249,174 TO 279,174

3108 POKE - 16302,0:
3110 HPLOT 4,175 TO 4,180: HPLOT
      46,175 TO 46,180: HPLOT 88,1
      75 TO 88,180: HPLOT 130,175 TO
      130,180: HPLOT 172,175 TO 17
      2,180
3111 HPLOT 214,175 TO 214,180: HPLOT
      270,175 TO 270,180
3115 FOR N = 1 TO 23 STEP 2
3116 VTAB (N): FOR K = 0 TO 39: PRINT
      " ";: NEXT K,N
3120 FOR N = 1 TO 10: READ G$: VTAB
      1 + 2 * N: PRINT G$: NEXT N
3130 PRINT : PRINT : PRINT "0
      2      4      6      8      10
      !";: POKE 2039,160: REM
      AVOID SCROLL
3190 POKE 36091,212
3200 CALL 35584: REM MIX FIELDS

3990 POKE 16368,0: REM RESET KE
      Y STROB
```

PROGRAM 5-2, CONT'D...

```
4000 REM *** BYE BYE ***

4005 PRINT : PRINT "BLOAD VFFS.B
      YE": REM CTRL D
4008 GR : TEXT
4009 : POKE - 16368,0: GR : TEXT

4010 HOME : VTAB 7: HTAB 20: PRINT
      "BYE";
4015 FOR N = 1 TO 1000: NEXT N
4020 HGR : HCOLOR= 5: HPLOT 143,
      86 TO 143,79: HPLOT 145,86 TO
      145,79: HPLOT 141,78 TO 141,
      77: HPLOT 139,78 TO 139,77: HPLOT
      147,78 TO 147,77: HPLOT 149,
      78 TO 149,77
4022 HPLOT 137,76 TO 137,73: HPLOT
      135,76 TO 135,73: HPLOT 151,
      76 TO 151,73: HPLOT 153,76 TO
      153,73
4024 HPLOT 107,73 TO 107,86: HPLOT
      109,73 TO 109,86: HPLOT 119,
      75 TO 119,78: HPLOT 121,75 TO
      121,78: HPLOT 119,81 TO 119,
      84: HPLOT 121,81 TO 121,84
4026 HPLOT 107,73 TO 117,73: HPLOT
      107,74 TO 117,74: HPLOT 107,
      79 TO 117,79: HPLOT 107,80 TO
      117,80: HPLOT 107,85 TO 117,
      85: HPLOT 107,86 TO 117,86
4027 HPLOT 163,73 TO 163,86: HPLOT
      165,73 TO 165,86: HPLOT 163,
      73 TO 179,73: HPLOT 163,74 TO
      179,74
4028 HPLOT 163,85 TO 179,85: HPLOT
      163,86 TO 179,86: HPLOT 163,
      79 TO 175,79: HPLOT 163,80 TO
      175,80
4029 COLOR= 0: FOR N = 25 TO 39:
      HLIN 0,39 AT N: NEXT N
4030 COLOR= 3: HLIN 11,14 AT 25:
      HLIN 11,14 AT 28: HLIN 11,1
      4 AT 31: VLIN 25,31 AT 11: VLIN
      26,27 AT 15: VLIN 29,30 AT 1
      5
4032 VLIN 25,26 AT 18: VLIN 27,2
      7 AT 19: VLIN 28,31 AT 20: VLIN
      27,27 AT 21: VLIN 25,26 AT 2
      2
4034 VLIN 25,31 AT 25: HLIN 25,2
      9 AT 25: HLIN 25,28 AT 28: HLIN
      25,29 AT 31
4099 POKE 36091,128
```

PROGRAM 5-2, CONT'D...

```
4100  CALL 35584: REM  MIX FIELDS

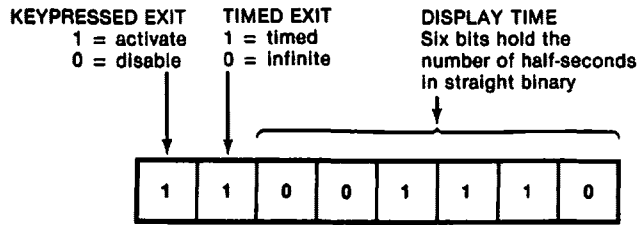
4200  PRINT : PRINT "RUN MENU": REM
      CTRL D
4201  REM  DELETE 4200 IF AUTO-ME
      NU IS NOT IN USE
4210  END

20000  REM  ** SNORGEL CURVES **

20005  HCOLOR= 3
20010  FOR X = 1 TO 200
20020  Y = (1 - (2.718) ^ ( - X /
      50))
20030  Z = Y * (2.718 ^ ( - X / 60
      ))
20040  HPLOT X + 61,159 - 137 * Y
20050  HPLOT X + 61,159 - 457 * Z

20060  NEXT X
29999  RETURN
30000  DATA  2,6,9,16,21,24,27,29
      ,31,39

30100  DATA  VELMA,GERTRUDE,CHARI
      TY,NOREEN, EMILY,PEGGY,ELAIN
      E,SAMANTHA,RACHAEL,YVONNE
```



The CONTROL file is presently located at hex \$8CFB (decimal 36091).

The values shown give a keypressed exit along with a timeout exit after seven seconds.

Some CONTROL file use examples:

1. Keypressed exit only—use hex \$80 or decimal 128.
2. Four-second display only—use hex \$48 or decimal 72.
3. Four-second display with keypressed exit—use hex \$C8 or decimal 200.
4. Eight-second display only—use hex \$50 or decimal 80.
5. Eight-second display with keypressed exit—use hex \$D0 or decimal 208.

To set up CONTROL in hex, start with \$00 and add \$80 if keypressed exit is wanted. Add \$40 if timeout is wanted. Add TWICE the number of seconds of timeout to this value (max. timeout value = \$3F). Store this value in \$8CFB.

To set up CONTROL in decimal, start with 0 and add 128 if keypressed exit is wanted. Add 64 if timeout is wanted. Add TWICE the number of seconds of timeout to this value (max. timeout value = 63). Poke this value into 36091.

Do not use a CONTROL value of hex \$00 or a decimal 0 as this will give a permanent display.

Fig. 5-1. CONTROL file used in VFFS.

bits hold a binary number equal to the number of *half seconds* you wish to display. For instance, for a ten-second display, you would load decimal twenty, or hex \$14, into these six bytes.

Timeout range goes from one-half a second to thirty-two seconds, with the option of an "infinite" timeout exited with a pressed key.

Next are the HPAT files, short for *Horizontal Patterns*. The following example shows us the HPAT addresses and data value. There are four possible horizontal pattern files used in VFFS. These files decide which soft switch is flipped in any of ten exact horizontal screen positions.

The files are called HPAT1 through HPAT4. The display always starts with HPAT1. Each HPAT file continues for a number of scan lines set by the VPAT file shown in Table 5-1. While the HPAT files must be sequenced in order, they can repeat as often as you like each field.

Here are the present addresses of the pattern files . . .

HPAT1 ADDRESSES		
Switch-on Character	Hex Address	Decimal Address
0	\$8B34	35636
4	\$8B37	35639
8	\$8B3A	35642
12	\$8B3D	35645
16	\$8B40	35648
20	\$8B43	35651
24	\$8B46	35654
28	\$8B49	35657
32	\$8B4C	35660
36	\$8B4F	35663

HPAT2 ADDRESSES		
Switch-on Character	Hex Address	Decimal Address
0	\$8B97	35735
4	\$8B9A	35738
8	\$8B9D	35741
12	\$8BA0	35744
16	\$8BA3	35747
20	\$8BA6	35750
24	\$8BA9	35753
28	\$8BAC	35756
32	\$8BAF	35759
36	\$8BB2	35762

HPAT3 ADDRESSES		
Switch-on Character	Hex Address	Decimal Address
0	\$8BC7	35783
4	\$8BCA	35786
8	\$8BCD	35789
12	\$8BD0	35792
16	\$8BD3	35795
20	\$8BD6	35798
24	\$8BD9	35801
28	\$8BDC	35804
32	\$8BDF	35807
36	\$8BE2	35810

HPAT4 ADDRESSES		
Switch-on Character	Hex Address	Decimal Address
0	\$8B64	35684
4	\$8B67	35687
8	\$8B6A	35690
12	\$8B6D	35693
16	\$8B70	35696
20	\$8B73	35699
24	\$8B76	35702
28	\$8B79	35705
32	\$8B7C	35708
36	\$8B7F	35711

Note that any address in the I/O space below \$C100 may also be flipped by using its hex address minus \$C000, or its decimal equivalent. Note, also, that any change or relocation of the VFFS program will change all these address locations.

HPAT is really working code rather than a true file, so its soft switching "file" values only show up every *third* byte. HPAT splits up the screen horizontally

Here are the data values used in an HPAT file. . . .

HPAT DATA VALUES		
Switch	Hex	Decimal
Graphics ON	\$50	80
Text ON	\$51	81
Full Screen	\$52	82
Mixed Graphics	\$53	83
Page ONE	\$54	84
Page TWO	\$55	85
LORES Graphics	\$56	86
HIRES Graphics	\$57	87
Do Nothing	\$60	96

into ten groups of four characters each. At the beginning of each four-character slot, you have the option of flipping one soft switch.

For instance, suppose we use this HPAT code . . .

```
8B33: 8D 60 CO STA DUMMY ; Flip dummy soft switch
8B36: 8D 50 CO STA TEXT   ; Switch to text
8B39: 8D 60 CO STA DUMMY ; Flip dummy soft switch
8B3C: 8D 60 CO STA DUMMY ; Flip dummy soft switch
8B3F: 8D 51 CO STA GRAFIX ; Switch to graphics

8B42: 8D 60 CO STA DUMMY ; Flip dummy soft switch
8B45: 8D 60 CO STA DUMMY ; Flip dummy soft switch
8B48: 8D 60 CO STA DUMMY ; Flip dummy soft switch
8B4B: 8D 60 CO STA DUMMY ; Flip dummy soft switch
8B4E: 8D 60 CO STA DUMMY ; Flip dummy soft switch
```

What this will do is display the first four characters as graphics, the next twelve as text, and the remainder of the line as graphics again. Which type of graphics and which page text is decided by how the *other* soft switches have been previously flipped.

Note that you only have to flip a soft switch when you want to produce a *change* in the output mode. If you flip to text, you stay that way till you flip to something else.

Each STA absolute store command takes 4 CPU clock cycles, which is equal to 4 characters. Thus, the 10 switch flippings will equal the horizontal live-character scan time of 40 CPU clock cycles.

To use this particular HPAT file, we access it with some machine-language code that goes through the "file" exactly once each horizontal line. The timing is very carefully set up so that the first switch flips before character Zero, the second one before character Four, the next one before character Eight, and so on, across the screen to the final switch that flips before character Thirty-six.

Because our back is to the wall with CPU operating speeds, we aren't free to flip only those switches we want to. Instead, we must flip 10 soft switches each

and every horizontal line. To do this, we fill in with *dummy* soft switches. A dummy soft switch is one that flips but doesn't do anything. One good dummy soft switch is a write to location \$C060. This is the cassette IN location, and the hardware here is read-only. So, you can write to location \$C060 all day long and nothing will happen. By coincidence, \$C060 is also read as part of the exact locking code. These two uses of one address space location will not cause conflict.

What you usually do is start with a VFFS.EMPTY file that holds all dummy soft switches. Then, you modify that file by changing the code locations you need to flip the "live" switches you really want in the exact slots where you want them. The easiest way to change these locations is to display the file in machine language and, then, use the monitor commands to modify only the locations you want. You can also change a VFFS file by poking from Applesoft, by reading Applesoft data statements, by transferring the commands from another machine-language file, or by reading a new "file" off your disk.

If we had only one HPAT file, we could only have one combination of text, HIRES, and LORES which would have to repeat each and every horizontal line all the way down the screen. At the other extreme, we could have 192 separate HPAT files that would let us change each horizontal line any way we liked, separate from all the others. But this, of course, would take bunches of code and would be horribly complex.

Instead, we have shown you four HPAT files, called HPAT1 through HPAT4. This lets you have four different horizontal patterns. We will see that we have an option of changing to the next HPAT file at any scan line we like. Thus, you could use HPAT1 for fifteen lines, HPAT2 for six lines, HPAT3 for 97 lines, and HPAT4 for two lines. If we like, we can go round and round back through the HPAT files as often as we want to on the same field. The next field always restarts with HPAT1.

Four different horizontal patterns that can be repeated is more than enough for text insets and some really fancy field mixing, yet it keeps our program and pattern files down to something manageable.

Our final file is called VPATRN. VPATRN does two things for you. It first lets you flip an eleventh soft switch during each horizontal blanking time. This extra switch can be handy in going, say from text page One to HIRES page Two. You can use this extra and hidden switch any way you want to. The second thing that VPATRN does for you is let you advance to the next available HPAT file any time that you like.

To repeat, the VPATRN file lets you flip one soft switch invisibly during the blanking time of each horizontal line. It also optionally lets you change to the next available HPAT horizontal pattern. A *cleared* MSB in a VPATRN file location only flips a soft switch, while a *set* MSB in the file both flips a soft switch and moves on to the next available HPAT file. Should no soft switching be wanted, a dummy or "do-nothing" switch location is substituted.

Table 5-1 gives the addresses of the VPATRN files

Table 5-1. VPATRN Vertical Pattern File Used in VFFS

Line number	Character and dot	LORES line number	Hex address	Decimal address
0	0/0	0/0	8CC0	36032
1	0/1	0/1	8CBF	36031
2	0/2	0/2	8CBE	36030
3	0/3	0/3	8CBD	36029

Table 5-1 Cont. VPATRN Vertical Pattern File Used in VFFS

Line number	Character and dot	LORES line number	Hex address	Decimal address
4	0/4	1/0	8CBC	36028
5	0/5	1/1	8CBB	36027
6	0/6	1/2	8CBA	36026
7	0/7	1/3	8CB9	36025
8	1/0	2/0	8CB8	36024
9	1/1	2/1	8CB7	36023
10	1/2	2/2	8CB6	36022
11	1/3	2/3	8CB5	36021
12	1/4	3/0	8CB4	36020
13	1/5	3/1	8CB3	36019
14	1/6	3/2	8CB2	36018
15	1/7	3/3	8CB1	36017
16	2/0	4/0	8CB0	36016
17	2/1	4/1	8CAF	36015
18	2/2	4/2	8CAE	36014
19	2/3	4/3	8CAD	36013
20	2/4	5/0	8CAC	36012
21	2/5	5/1	8CAB	36011
22	2/6	5/2	8CAA	36010
23	2/7	5/3	8CA9	36009
24	3/0	6/0	8CA8	36008
25	3/1	6/1	8CA7	36007
26	3/2	6/2	8CA6	36006
27	3/3	6/3	8CA5	36005
28	3/4	7/0	8CA4	36004
29	3/5	7/1	8CA3	36003
30	3/6	7/2	8CA2	36002
31	3/7	7/3	8CA1	36001
32	4/0	8/0	8CA0	36000
33	4/1	8/1	8C9F	35999
34	4/2	8/2	8C9E	35998
35	4/3	8/3	8C9D	35997
36	4/4	9/0	8C9C	35996
37	4/5	9/1	8C9B	35995
38	4/6	9/2	8C9A	35994
39	4/7	9/3	8C99	35993
40	5/0	10/0	8C98	35992
41	5/1	10/1	8C97	35991
42	5/2	10/2	8C96	35990
43	5/3	10/3	8C95	35989
44	5/4	11/0	8C94	35988
45	5/5	11/1	8C93	35987
46	5/6	11/2	8C92	35986
47	5/7	11/3	8C91	35985
48	6/0	12/0	8C90	35984
49	6/1	12/1	8C8F	35983
50	6/2	12/2	8C8E	35982
51	6/3	12/3	8C8D	35981
52	6/4	13/0	8C8C	35980
53	6/5	13/1	8C8B	35979
54	6/6	13/2	8C8A	35978
55	6/7	13/3	8C89	35977

Table 5-1 Cont. VPATRN Vertical Pattern File Used in VFFS

Line number	Character and dot	LORES line number	Hex address	Decimal address
56	7/0	14/0	8C88	35976
57	7/1	14/1	8C87	35975
58	7/2	14/2	8C86	35974
59	7/3	14/3	8C85	35973
60	7/4	15/0	8C84	35972
61	7/5	15/1	8C83	35971
62	7/6	15/2	8C82	35970
63	7/7	15/3	8C81	35969
64	8/0	16/0	8C80	35968
65	8/1	16/1	8C7F	35967
66	8/2	16/2	8C7E	35966
67	8/3	16/3	8C7D	35965
68	8/4	17/0	8C7C	35964
69	8/5	17/1	8C7B	35963
70	8/6	17/2	8C7A	35962
71	8/7	17/3	8C79	35961
72	9/0	18/0	8C78	35960
73	9/1	18/1	8C77	35959
74	9/2	18/2	8C76	35958
75	9/3	18/3	8C75	35957
76	9/4	19/0	8C74	35956
77	9/5	19/1	8C73	35955
78	9/6	19/2	8C72	35954
79	9/7	19/3	8C71	35953
80	10/0	20/0	8C70	35952
81	10/1	20/1	8C6F	35951
82	10/2	20/2	8C6E	35950
83	10/3	20/3	8C6D	35949
84	10/4	21/0	8C6C	35948
85	10/5	21/1	8C6B	35947
86	10/6	21/2	8C6A	35946
87	10/7	21/3	8C69	35945
88	11/0	22/0	8C68	35944
89	11/1	22/1	8C67	35943
90	11/2	22/2	8C66	35942
91	11/3	22/3	8C65	35941
92	11/4	23/0	8C64	35940
93	11/5	23/1	8C63	35939
94	11/6	23/2	8C62	35938
95	11/7	23/3	8C61	35937
96	12/0	24/0	8C60	35936
97	12/1	24/1	8C5F	35935
98	12/2	24/2	8C5E	35934
99	12/3	24/3	8C5D	35933
100	12/4	25/0	8C5C	35932
101	12/5	25/1	8C5B	35931
102	12/6	25/2	8C5A	35930
103	12/7	25/3	8C59	35929
104	13/0	26/0	8C58	35928
105	13/1	26/1	8C57	35927
106	13/2	26/2	8C56	35926
107	13/3	26/3	8C55	35925

Table 5-1 Cont. VPATRN Vertical Pattern File Used in VFFS

Line number	Character and dot	LORES line number	Hex address	Decimal address
108	13/4	27/0	8C54	35924
109	13/5	27/1	8C53	35923
110	13/6	27/2	8C52	35922
111	13/7	27/3	8C51	35921
112	14/0	28/0	8C50	35920
113	14/1	28/1	8C4F	35919
114	14/2	28/2	8C4E	35918
115	14/3	28/3	8C4D	35917
116	14/4	29/0	8C4C	35916
117	14/5	29/1	8C4B	35915
118	14/6	29/2	8C4A	35914
119	14/7	29/3	8C49	35913
120	15/0	30/0	8C48	35912
121	15/1	30/1	8C47	35911
122	15/2	30/2	8C46	35910
123	15/3	30/3	8C45	35909
124	15/4	31/0	8C44	35908
125	15/5	31/1	8C43	35907
126	15/6	31/2	8C42	35906
127	15/7	31/3	8C41	35905
128	16/0	32/0	8C40	35904
129	16/1	32/1	8C3F	35903
130	16/2	32/2	8C3E	35902
131	16/3	32/3	8C3D	35901
132	16/4	33/0	8C3C	35900
133	16/5	33/1	8C3B	35899
134	16/6	33/2	8C3A	35898
135	16/7	33/3	8C39	35897
136	17/0	34/0	8C38	35896
137	17/1	34/1	8C37	35895
138	17/2	34/2	8C36	35894
139	17/3	34/3	8C35	35893
140	17/4	35/0	8C34	35892
141	17/5	35/1	8C33	35891
142	17/6	35/2	8C32	35890
143	17/7	35/3	8C31	35889
144	18/0	36/0	8C30	35888
145	18/1	36/1	8C2F	35887
146	18/2	36/2	8C2E	35886
147	18/3	36/3	8C2D	35885
148	18/4	36/4	8C2C	35884
149	18/5	36/5	8C2B	35883
150	18/6	36/6	8C2A	35882
151	18/7	36/7	8C29	35881
152	19/0	38/0	8C28	35880
153	19/1	38/1	8C27	35879
154	19/2	38/2	8C26	35878
155	19/3	38/3	8C25	35877
156	19/4	39/0	8C24	35876
157	19/5	39/1	8C23	35875
158	19/6	39/2	8C22	35874
159	19/7	39/3	8C21	35873

Table 5-1 Cont. VPATRN Vertical Pattern File Used in VFFS

Line number	Character and dot	LORES line number	Hex address	Decimal address
160	20/0	40/0	8C20	35872
161	20/1	40/1	8C1F	35871
162	20/2	40/2	8C1E	35870
163	20/3	40/3	8C1D	35869
164	20/4	41/0	8C1C	35868
165	20/5	41/1	8C1B	35867
166	20/6	41/2	8C1A	35866
167	20/7	41/3	8C19	35865
168	21/0	42/0	8C18	35864
169	21/1	42/1	8C17	35863
170	21/2	42/2	8C16	35862
171	21/3	42/3	8C15	35861
172	21/4	43/0	8C14	35860
173	21/5	43/1	8C13	35859
174	21/6	43/2	8C12	35858
175	21/7	43/3	8C11	35857
176	22/0	44/0	8C10	35856
177	22/1	44/1	8C0F	35855
178	22/2	44/2	8C0E	35854
179	22/3	44/3	8C0D	35853
180	22/4	45/0	8C0C	35852
181	22/5	45/1	8C0B	35851
182	22/6	45/2	8C0A	35850
183	22/7	45/3	8C09	35849
184	23/0	46/0	8C08	35848
185	23/1	46/1	8C07	35847
186	23/2	46/2	8C06	35846
187	23/3	46/3	8C05	35845
188	23/4	47/0	8C04	35844
189	23/5	47/1	8C03	35843
190	23/6	47/2	8C02	35842
191	23/7	47/3	8C01	35841

Note that any soft switch in the I/O space below C080 may be flipped. Use its hex address minus \$C000 if a new HPAT is *not* to be used, or its hex address minus \$BF80 if a new HPAT *is* to be used. Note also that any change or relocation of the VFFS program will change all these address locations.

There are 192 entries in the VPATRN file, arranged from the bottom to the top of the screen. These entries start out as \$60 values which end up stored at \$C060 as dummy writes to the cassette IN read-only location. Changing a value to \$54, for instance, causes a soft switch to flip to page One. A \$55 flips you to page Two, and so on. Or, for far out uses, a \$59 will set annunciator AN0 at the start of a chosen vertical line, and a \$58 will clear AN0.

If a VPATRN file entry has its most significant bit set, it will flip the intended soft switch and automatically advance us to the next available HPAT file. If the VPATRN entry has its most significant bit cleared, it will flip the intended soft switch but will not change the HPAT file in use.

For instance, a \$51 entry will flip you to text and keep the old HPAT in use. A \$D1 will both flip you to text and advance to the next HPAT. Note that \$51 + \$80 = \$D1. If the most significant VPATRN file bit is *set*, then HPAT gets *flipped*. If the MSB is *cleared*, then HPAT *stays* the same.

The following are the data values used in a VPATRN file

Switch	Advance HPAT?	Hex Data	Decimal Data
Do Nothing	no	\$60	96
	yes	\$E0	224
Graphics ON	no	\$50	80
	yes	\$D0	208
Text ON	no	\$51	81
	yes	\$D1	209
Full Screen	no	\$52	82
	yes	\$D2	210
Mixed Graphics	no	\$53	83
	yes	\$D3	211
Page One	no	\$54	84
	yes	\$D4	212
Page Two	no	\$55	85
	yes	\$D5	213
LORES Graphics	no	\$56	86
	yes	\$D6	214
HIRES Graphics	no	\$57	87
	yes	\$D7	215

The main VFFS program is set up to always start on HPAT1 at the top of the screen, and continue from there. Values in the VPATRN file will decide when and if a change to HPAT2, HPAT3, and so on, is to be done. If you are using HPAT4 and you get a change command, you switch back to HPAT1 and go round and round.

VFFS DETAILS

Note that all these CONTROL, HPAT, and VPATRN file locations will change if you relocate the program or make any other modifications to it.

What you do ahead of time is make up a VFFS.EMPTY program with all dummy soft switches in VPATRN and the four HPAT files, along with some sensible value for CONTROL, say a four-second display with keypressed active. You then modify these files to do the exact display job you want. For many uses, only a few file locations need be changed.

A pair of worksheets that make file design a lot easier appear in Figs. 5-2 and 5-3. The easiest way I've found to change an HPAT or VPATRN file is to simply list it in machine language on the screen, and then change the dummy locations as needed. You then save VFFS under a new modifier name to pick up these special locations.

To use these worksheets, first split your display up into horizontal blocks, each of which has a distinct horizontal pattern. Then, write the modes into the top of the HPAT file worksheet, putting a "TEXT" in the block where you want text to *start* appearing, and so on. Then, put the right data value in the right box on the correct HPAT for each switch flipping. To finish your HPAT worksheet, calculate the line number or numbers where you want to switch from HPAT1 to HPAT2, and so on. Put these numbers in the bottom boxes.

Now go to the VPATRN worksheet, and label the changes you want in the positions you want. Then, substitute the code needed beside each position. Note that you can count in scan lines, in 8 scan-line character units, or in 4 scan-line LORES units.

When you have completed both worksheets, you should have a list of all file values that you need. Load VFFS.EMPTY, modify these file values, change CONTROL as needed, and resave your new VFFS under a new name. The whole process is much easier to do than to describe.

Let's look at some more details of VFFS.EMPTY that appear as Program 5-1. A flowchart is shown in Fig. 5-4.

To activate VFFS, you first create your own custom version and then load it into your Apple. Then, you do a JSR \$8B00 or a CALL 35584. Your VFFS will then do a mixed field display for you for the length of time that you selected with your CONTROL data values.

At the start of VFFS, your Apple is still displaying whatever it happened to have on the screen. Nothing visible will change until an exact lock to video timing is completed. Once again, you must have the one-wire sync mod of the previous enhancement in place for mixed fields to work.

We first do some setting up at \$8B00. This takes the timeout bits from CONTROL and moves them to a location called TIMER where they can be counted down for timeout. We then find the exact start of a video field using code we borrowed from EXACTF of the last enhancement. This code must have the field sync modification of Enhancement 4 to work. What EXACTF does is find the beginning of a field with some jitter and, then, backs up one CPU clock cycle per field till it finds an exact field start.

We get to \$8B24 exactly at the start of a field. We call this location NEWFLD since it is the point we will return to after each field is complete. Timing from now on is exactly controlled to take up precisely one field per trip around. Up to this point, the Apple is still displaying whatever it happened to be showing before we began.

NEWFLD tells us the number of live scan lines we are to use. This is usually hex \$C0 or decimal 192. The number of live scan lines is put in the Y register and Y counts the lines down for us. We then grab a value from the VPATRN file for the top line. This is done with an indexed load, taking the sum of the start of the VATRN file at \$8C00 and the value in Y. Thus, our top line appears as VPATRN byte \$8CC0.

We next test the value found at \$8CC0 to see if we are to flip to the next HPAT. If \$8CC0 has its MSB cleared, we use HPAT1; if the MSB is set, we switch to HPAT2. This is done with the BMI test at \$8B29.

Assuming that we don't want to change HPAT just yet, we go on through the code starting at \$8B2F. We first flip the VPATRN byte switch invisibly during the blanking time and, then, go on to switch up to ten locations set by the HPAT1 file.

We reach the right end of the screen on the first scan line when we get to \$8B51. We subtract one line from our scan counter and repeat the process, taking exactly 65 CPU cycles to get right back where we started from, but one horizontal scan line down.

This goes on and on until we either reach the bottom of the screen, or else, until we find a VATRN value that flips to us to HPAT2. The time it takes to switch *between* HPAT files is exactly the same as the time it takes to *continue* with the same HPAT file. The apparent order of the HPAT files is "juggled" in the program to keep all the relative branches from pattern to pattern in range.

The process repeats every scan line. A VPATRN byte is gotten and tested to see if a switch to the next HPAT is needed. Then, the chosen HPAT scan is done,

Fig. 5-2. Horizontal pattern VFFS worksheet.

HORIZONTAL PATTERN WORKSHEET FOR VFFS.

ACTION POINTS

II
III
IV
V
VI
VII
VIII
IX
X

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
---	---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

HORIZONTAL
CHARACTER
POSITION

HPAT1

HPAT2

HPAT3

HPAT4

\$8B34 I

\$8B37 II

\$8B3A III

\$8B3D IV

\$8B40 V

\$8B43 VI

\$8B46 VII

\$8B49 VIII

\$8B4C IX

\$8B4F X

**HPAT1
STARTS
ON LINES**

0,

\$8B97 I

\$8B9A II

\$8B9D III

\$8BA0 IV

\$8BA3 V

\$8BA6 VI

\$8BA9 VII

\$8BAC VIII

\$8BAF IX

\$8BB2 X

**HPAT2
STARTS
ON LINES**

,

\$8BC7 I

\$8BCA II

\$8BCD III

\$8BD0 IV

\$8BD3 V

\$8BD6 VI

\$8BD9 VII

\$8BDC VIII

\$8BDF IX

\$8BE2 X

**HPAT3
START
ON LINES**

,

\$8B64 I

\$8B67 II

\$8B6A III

\$8B6D IV

\$8B70 V

\$8B73 VI

\$8B76 VII

\$8B79 VIII

\$8B7C IX

\$8B7F X

**HPAT4
STARTS
ON LINES**

,

DATA VALUES	
GRAPHICS	50
TEXT	51
FULL	52
MIXED	53
PAGE 1	54
PAGE 2	55
LORES	56
HIRES	57
DUMMY	60
	
	
	
	
VALUES IN \$HEX	

VERTICAL PATTERN WORKSHEET FOR VFFS.											
CHARACTER 0		CHARACTER 6		CHARACTER 12		CHARACTER 18					
0	<input type="checkbox"/>	\$8C0	48	<input type="checkbox"/>	\$8C90	96	<input type="checkbox"/>	\$8C60	144	<input type="checkbox"/>	\$8C30
1	<input type="checkbox"/>	\$8CBF	49	<input type="checkbox"/>	\$8C8F	97	<input type="checkbox"/>	\$8C5F	145	<input type="checkbox"/>	\$8C2F
2	<input type="checkbox"/>	\$8CBE	50	<input type="checkbox"/>	\$8C8E	98	<input type="checkbox"/>	\$8C5E	146	<input type="checkbox"/>	\$8C2E
3	<input type="checkbox"/>	\$8CBD	51	<input type="checkbox"/>	\$8C8D	99	<input type="checkbox"/>	\$8C5D	147	<input type="checkbox"/>	\$8C2D
4	<input type="checkbox"/>	\$8CBC	52	<input type="checkbox"/>	\$8C8C	100	<input type="checkbox"/>	\$8C5C	148	<input type="checkbox"/>	\$8C2C
5	<input type="checkbox"/>	\$8CBB	53	<input type="checkbox"/>	\$8C8B	101	<input type="checkbox"/>	\$8C5B	149	<input type="checkbox"/>	\$8C2B
6	<input type="checkbox"/>	\$8CBA	54	<input type="checkbox"/>	\$8C8A	102	<input type="checkbox"/>	\$8C5A	150	<input type="checkbox"/>	\$8C2A
7	<input type="checkbox"/>	\$8CB9	55	<input type="checkbox"/>	\$8C89	103	<input type="checkbox"/>	\$8C59	151	<input type="checkbox"/>	\$8C29
LORES 0/1		LORES 12/13		LORES 24/25		LORES 36/37					
CHARACTER 1		CHARACTER 7		CHARACTER 13		CHARACTER 19					
8	<input type="checkbox"/>	\$8CB8	56	<input type="checkbox"/>	\$8C88	104	<input type="checkbox"/>	\$8C58	152	<input type="checkbox"/>	\$8C28
9	<input type="checkbox"/>	\$8CB7	57	<input type="checkbox"/>	\$8C87	105	<input type="checkbox"/>	\$8C57	153	<input type="checkbox"/>	\$8C27
10	<input type="checkbox"/>	\$8CB6	58	<input type="checkbox"/>	\$8C86	106	<input type="checkbox"/>	\$8C56	154	<input type="checkbox"/>	\$8C26
11	<input type="checkbox"/>	\$8CB5	59	<input type="checkbox"/>	\$8C85	107	<input type="checkbox"/>	\$8C55	155	<input type="checkbox"/>	\$8C25
12	<input type="checkbox"/>	\$8CB4	60	<input type="checkbox"/>	\$8C84	108	<input type="checkbox"/>	\$8C54	156	<input type="checkbox"/>	\$8C24
13	<input type="checkbox"/>	\$8CB3	61	<input type="checkbox"/>	\$8C83	109	<input type="checkbox"/>	\$8C53	157	<input type="checkbox"/>	\$8C23
14	<input type="checkbox"/>	\$8CB2	62	<input type="checkbox"/>	\$8C82	110	<input type="checkbox"/>	\$8C52	158	<input type="checkbox"/>	\$8C22
15	<input type="checkbox"/>	\$8CB1	63	<input type="checkbox"/>	\$8C81	111	<input type="checkbox"/>	\$8C51	159	<input type="checkbox"/>	\$8C21
LORES 2/3		LORES 14/15		LORES 26/27		LORES 38/39					
CHARACTER 2		CHARACTER 8		CHARACTER 14		CHARACTER 20					
16	<input type="checkbox"/>	\$8CB0	64	<input type="checkbox"/>	\$8C80	112	<input type="checkbox"/>	\$8C50	160	<input type="checkbox"/>	\$8C20
17	<input type="checkbox"/>	\$8CAF	65	<input type="checkbox"/>	\$8C7F	113	<input type="checkbox"/>	\$8C4F	161	<input type="checkbox"/>	\$8C1F
18	<input type="checkbox"/>	\$8CAE	66	<input type="checkbox"/>	\$8C7E	114	<input type="checkbox"/>	\$8C4E	162	<input type="checkbox"/>	\$8C1E
19	<input type="checkbox"/>	\$8CAD	67	<input type="checkbox"/>	\$8C7D	115	<input type="checkbox"/>	\$8C4D	163	<input type="checkbox"/>	\$8C1D
20	<input type="checkbox"/>	\$8CAC	68	<input type="checkbox"/>	\$8C7C	116	<input type="checkbox"/>	\$8C4C	164	<input type="checkbox"/>	\$8C1C
21	<input type="checkbox"/>	\$8CAB	69	<input type="checkbox"/>	\$8C7B	117	<input type="checkbox"/>	\$8C4B	165	<input type="checkbox"/>	\$8C1B
22	<input type="checkbox"/>	\$8CAA	70	<input type="checkbox"/>	\$8C7A	118	<input type="checkbox"/>	\$8C4A	166	<input type="checkbox"/>	\$8C1A
23	<input type="checkbox"/>	\$8CA9	71	<input type="checkbox"/>	\$8C79	119	<input type="checkbox"/>	\$8C49	167	<input type="checkbox"/>	\$8C19
LORES 4/5		LORES 16/17		LORES 28/29		LORES 40/41					
CHARACTER 3		CHARACTER 9		CHARACTER 15		CHARACTER 21					
24	<input type="checkbox"/>	\$8CA8	72	<input type="checkbox"/>	\$8C78	120	<input type="checkbox"/>	\$8C48	168	<input type="checkbox"/>	\$8C18
25	<input type="checkbox"/>	\$8CA7	73	<input type="checkbox"/>	\$8C77	121	<input type="checkbox"/>	\$8C47	169	<input type="checkbox"/>	\$8C17
26	<input type="checkbox"/>	\$8CA6	74	<input type="checkbox"/>	\$8C76	122	<input type="checkbox"/>	\$8C46	170	<input type="checkbox"/>	\$8C16
27	<input type="checkbox"/>	\$8CA5	75	<input type="checkbox"/>	\$8C75	123	<input type="checkbox"/>	\$8C45	171	<input type="checkbox"/>	\$8C15
28	<input type="checkbox"/>	\$8CA4	76	<input type="checkbox"/>	\$8C74	124	<input type="checkbox"/>	\$8C44	172	<input type="checkbox"/>	\$8C14
29	<input type="checkbox"/>	\$8CA3	77	<input type="checkbox"/>	\$8C73	125	<input type="checkbox"/>	\$8C43	173	<input type="checkbox"/>	\$8C13
30	<input type="checkbox"/>	\$8CA2	78	<input type="checkbox"/>	\$8C72	126	<input type="checkbox"/>	\$8C42	174	<input type="checkbox"/>	\$8C12
31	<input type="checkbox"/>	\$8CA1	79	<input type="checkbox"/>	\$8C71	127	<input type="checkbox"/>	\$8C41	175	<input type="checkbox"/>	\$8C11
LORES 6/7		LORES 18/19		LORES 30/31		LORES 42/43					
CHARACTER 4		CHARACTER 10		CHARACTER 16		CHARACTER 22					
32	<input type="checkbox"/>	\$8CA0	80	<input type="checkbox"/>	\$8C70	128	<input type="checkbox"/>	\$8C40	176	<input type="checkbox"/>	\$8C10
33	<input type="checkbox"/>	\$8C9F	81	<input type="checkbox"/>	\$8C6F	129	<input type="checkbox"/>	\$8C3F	177	<input type="checkbox"/>	\$8C0F
34	<input type="checkbox"/>	\$8C9E	82	<input type="checkbox"/>	\$8C6E	130	<input type="checkbox"/>	\$8C3E	178	<input type="checkbox"/>	\$8C0E
35	<input type="checkbox"/>	\$8C9D	83	<input type="checkbox"/>	\$8C6D	131	<input type="checkbox"/>	\$8C3D	179	<input type="checkbox"/>	\$8C0D
36	<input type="checkbox"/>	\$8C9C	84	<input type="checkbox"/>	\$8C6C	132	<input type="checkbox"/>	\$8C3C	180	<input type="checkbox"/>	\$8C0C
37	<input type="checkbox"/>	\$8C9B	85	<input type="checkbox"/>	\$8C6B	133	<input type="checkbox"/>	\$8C3B	181	<input type="checkbox"/>	\$8C0B
38	<input type="checkbox"/>	\$8C9A	86	<input type="checkbox"/>	\$8C6A	134	<input type="checkbox"/>	\$8C3A	182	<input type="checkbox"/>	\$8C0A
39	<input type="checkbox"/>	\$8C99	87	<input type="checkbox"/>	\$8C69	135	<input type="checkbox"/>	\$8C39	183	<input type="checkbox"/>	\$8C09
LORES 8/9		LORES 20/21		LORES 32/33		LORES 44/45					
CHARACTER 5		CHARACTER 11		CHARACTER 17		CHARACTER 23					
40	<input type="checkbox"/>	\$8C98	88	<input type="checkbox"/>	\$8C68	136	<input type="checkbox"/>	\$8C38	184	<input type="checkbox"/>	\$8C08
41	<input type="checkbox"/>	\$8C97	89	<input type="checkbox"/>	\$8C67	137	<input type="checkbox"/>	\$8C37	185	<input type="checkbox"/>	\$8C07
42	<input type="checkbox"/>	\$8C96	90	<input type="checkbox"/>	\$8C66	138	<input type="checkbox"/>	\$8C36	186	<input type="checkbox"/>	\$8C06
43	<input type="checkbox"/>	\$8C95	91	<input type="checkbox"/>	\$8C65	139	<input type="checkbox"/>	\$8C35	187	<input type="checkbox"/>	\$8C05
44	<input type="checkbox"/>	\$8C94	92	<input type="checkbox"/>	\$8C64	140	<input type="checkbox"/>	\$8C34	188	<input type="checkbox"/>	\$8C04
45	<input type="checkbox"/>	\$8C93	93	<input type="checkbox"/>	\$8C63	141	<input type="checkbox"/>	\$8C33	189	<input type="checkbox"/>	\$8C03
46	<input type="checkbox"/>	\$8C92	94	<input type="checkbox"/>	\$8C62	142	<input type="checkbox"/>	\$8C32	190	<input type="checkbox"/>	\$8C02
47	<input type="checkbox"/>	\$8C91	95	<input type="checkbox"/>	\$8C61	143	<input type="checkbox"/>	\$8C31	191	<input type="checkbox"/>	\$8C01
LORES 10/11		LORES 22/23		LORES 34/35		LORES 46/47					
DATA VALUES											
	GRAPHICS	TEXT	FULL	MIXED	PAGE 1	PAGE 2	LORES	HIRES	DUMMY		
SAME HPAT	\$50	\$51	\$52	\$53	\$54	\$55	\$56	\$57	\$60	SAME HPAT	
NEW HPAT	\$D0	\$D1	\$D2	\$D3	\$D4	\$D5	\$D6	\$D7	\$E0	NEW HPAT	

Fig. 5-3. Vertical pattern VFFS worksheet.

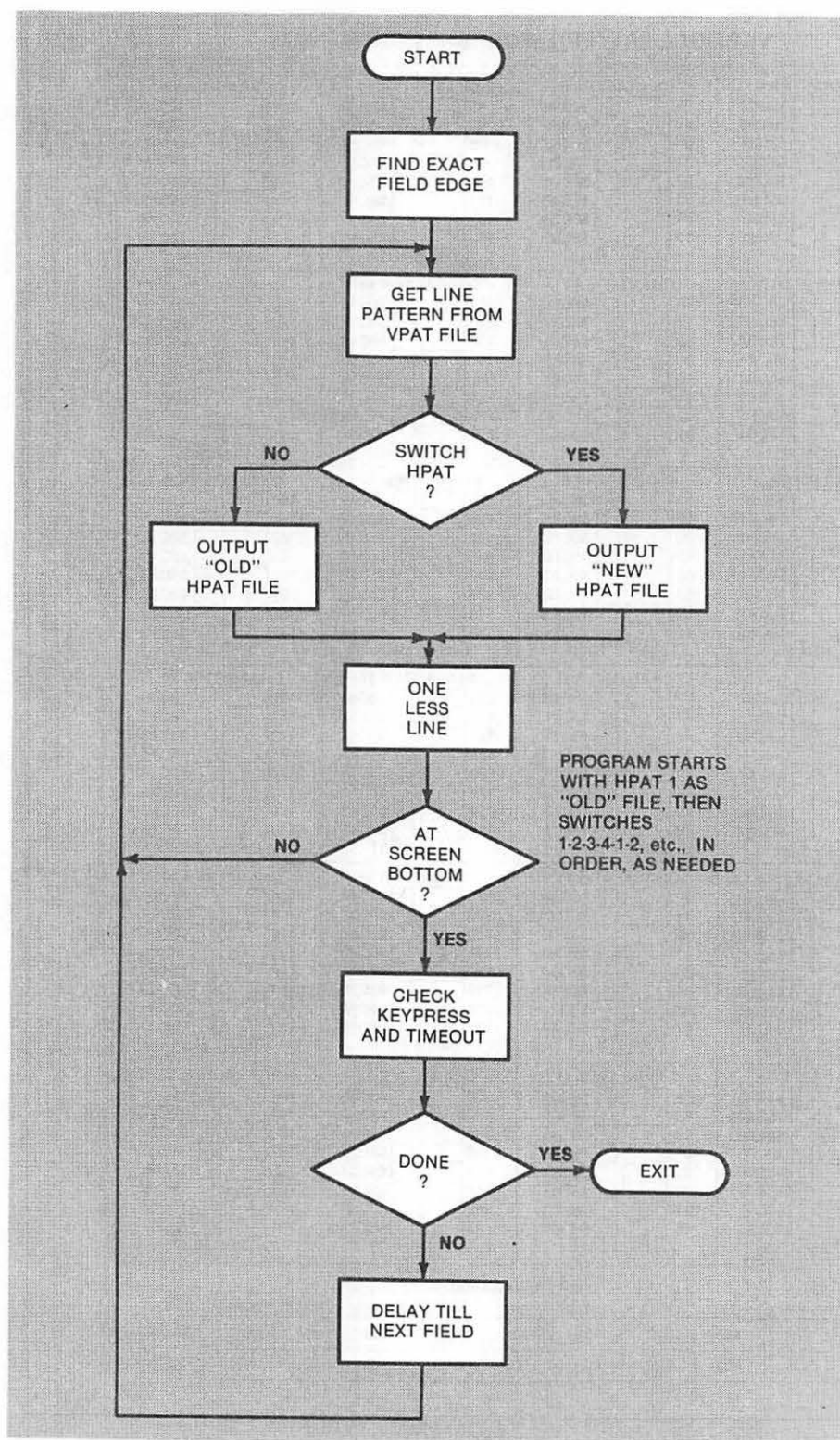


Fig. 5-4. Flowchart of VFFS.

flipping switches across the live part of the screen. During horizontal retrace, we knock one off the line number and get a new VPATRN value.

Eventually, we reach the bottom of the screen. Our test of the line number fails, and we drop first to location BOTTOM and, then, jump to BOTTM1. This absolute jump is needed because we go out of the allowed range for a relative jump. It is also more flexible this way, since you can substitute your own code for the entire vertical blanking interval.

Our default processing for the vertical blanking time starts at \$8CC2. We first see what the keypressed and timeout sub is up to. This sub is called KEYTIME.

KEYTIME first checks to see whether we are interested in a keypressed and then checks for an actual keypressed. If we want a key exit and if a key has been pressed, we return to the calling program in whatever language it was in. If not, we continue.

KEYTIME then increments a location called TIMEX. TIMEX is a multiplier that multiplies each count of the timeout value in CONTROL by 32. Thus, we only decrement the "half-seconds" count in TIMER only once every thirty-second field or, roughly, once each half second.

A check is then made to see if timeout is active, and if we are in the magic 1-of-32 field that lets us decrement the timeout counter. Should we timeout, we exit to the main program. If not, we go back to our BOTTM1 processing.

Note two things. First, the code in KEYTIME seems a little complex, but it has to be done this way so that each and every possible route through the code, that returns us to a repeat field scan, takes up *exactly* the same time. Anything else you do during the vertical blanking interval must also be set up to exactly take a precise and known number of CPU clock cycles for each and every possible direction through the code.

Secondly, note we have two different types of subroutine returns. If we simply RTS, then we kick back into our calling program, which is the BOTTM1 code of VFFS. If we pop the stack twice, removing the return address to VFFS, then we return to the main calling program in whatever language it happens to be. So, do an RTS to get back to VFFS, or else, a PLA-PLA-RTS to get back to your main calling program.

Assuming we haven't pressed a key or used up our timeout, we will normally return to BOTTM1. At this point, we have to delay exactly 4472 CPU cycles to get to the exact start of a new field. This is handled by a subroutine called VBSTAL. VBSTAL provides our vertical blanking delay for us. It does this by calling the WAIT delay subroutine in the monitor at \$FCA8 three times over, with just the right magic numbers stuffed into the accumulator to give us exactly the delay you need.

If you like, you can replace VBSTAL with any other program that takes exactly 4472 CPU clock cycles to repeat. This can give you a limited amount of animation or key entry while your mixed field display is active. Once again, the exact number of cycles used is critical and must be the same for each and every path through your code.

When VBSTAL is complete, we jump to NEWFLD and repeat the whole game over and over again till our timeout is complete or a selected key is pressed.

Your VFFS subroutine can be relocated to any protected space in memory, but note that all the file locations will change and all the worksheet values will be wrong if you try this.

As a hint of some of the really mind-boggling stuff you can do with mixed fields, think about what happens if you modify your files while they are running. This lets you do video wipes and other animation, where the screen modes change dynamically.

Heavy.

GLITCH RIDDANCE

The first few times you try to use the VFFS subs, you probably will get some ugly glitches messing up your display.

Your most obvious glitch source is caused by you not having what you think you do in your VPATRn and HPAT files. Be very careful and very patient in deciding what you want to put where on the screen. Use the worksheets. Note particularly that your HPAT values go every *third* byte in the program. When you list an HPAT, the HPAT values usually are placed in the slot *one byte higher* than the address shown on the screen.

For instance, if you have an

```
$8B30- 8D 60 C0      STA $C060
```

on your HPAT listing, it is location \$8B31 that holds your soft switch value, and NOT location \$8B30. The first byte of this code says to absolute store something. The second byte tells us the exact location on a particular memory page, and the final byte tells us exactly which page. Your HPAT value must fit into the middle slot and not either of the others. Mix these up and your program bombs for sure.

It is also a good idea to use every early slot left over on HPAT1 to reaffirm that your display is showing what you think it should be. As an example, you might want to flip the switches for page One, LORES, and full-screen display even if you never get out of these modes. That way, if you enter from some other part of a program or with a screen in the wrong mode, things will correct themselves immediately.

There are two more sources of glitches. Any time you flip into or out of LORES on screen, you will probably get the darndest half-and-half split HIRES/text glitch you ever saw. This happens if you are using a Revision No. 1 or newer Apple and it is caused by a switching change that Apple made to pick up extra HIRES colors. You can reduce this ugly glitch to one that can easily be handled by adding another modification to your Apple.

This modification is called a *Glitch Stomper* and is detailed in Enhancement 6. Your Glitch Stomper can be built for \$3.00 and in a few minutes time, and it is easy to add to your Apple.

The final glitch problem comes about since there is always one character "in the pipe" being processed, at the instant that you flip a soft switch . . .

GLITCH KILLING RULE
There is always one character or byte "in the pipe" at the instant that you flip a soft switch.

Say you flip from text to HIRES. What this rule says is that the next text character *after* the switch flips gets displayed as HIRES lines rather than dots. Or, going backwards, a flip from HIRES to text says that the HIRES dot pattern of the next byte *after* the switch flipping gets put on the display as a dot matrix character.

Now, that sounds just awful. But, you can carefully make each and every glitch on the screen invisible with some care. You can even make a glitch work for you, rather than against you.

For instance, when you go from text to LORES, exit to LORES black. If the text and LORES are on the same display page, the black LORES character has the same code as a text inverse @.

If you go from text to HIRES, if the next text character after the last live one is the blank character, you will get a vertical green line on the screen as your first HIRES display byte. What is happening is that the least significant seven bits of your ASCII text character, following the last valid text character, are displayed as HIRES. A text blank has the binary code 1010 0000. The most significant bit tells us to shift to HIRES color set "1," and the remaining "1" gives us a green line.

When you are building a HIRES graph, a green vertical line may be exactly what you want, and we get this "free." You can get most any other HIRES pattern in this slot that you want, as long as all eight horizontal scan lines have the same pattern on them. If you force a \$80 into the character slot following the last valid one, you will get both a black HIRES display and an invisible glitch.

If you go from HIRES to text on the live part of the scan, you will get an inverse @ as your first text character if the HIRES display was black at this point. This is caused by black being all zeros in the HIRES space and an inverse @ in the text space. The way to get rid of your inverse @'s is to change the code on the HIRES page to give you one vertical line through dot number 6 of the character. This will exit you to an ASCII code of blank.

Any other time that a glitch rears its ugly head, find out what character or byte is in which memory location. Then, find out what that character or byte looks like in the code that you just switched to. Then, change the character to make it invisible or otherwise useful.

Another obvious way to eliminate glitches is to use the VPATRN switches as much as possible. These switch during the horizontal blanking time and, therefore, any funny stuff will be off screen.

Glitch-riddance rules for very old "Revision 0" Apples will be slightly more complicated since the switching will take place one dot into the character. Have fun.

Summing up . . .

Glitches happen only during live scan switching.

The Glitch Stomper of the next enhancement converts really bad LORES glitches into workable ones.

By carefully studying where the code comes from and where it is going, all glitches can be made invisible, or else they can be made to work for you.

Some practice is all that it takes to build yourself some completely glitch-free displays that will mix and match text, HIRES, and LORES in any way that you like.

VFFS. EMPTY			
8AFF-	EA		
8B00-	20 FE 8C 2C 60 C0 10 FB		
8B08-	2C 60 C0 30 FB EA 10 00		
8B10-	A9 3B 20 A8 FC A9 34 20		
8B18-	A8 FC A9 01 20 A8 FC 2C		
8B20-	60 C0 10 EC A0 C0 B9 00		
8B28-	8C 30 63 10 00 29 7F AA		
8B30-	9D 00 C0 8D 60 C0 8D 60		
8B38-	C0 8D 60 C0 8D 60 C0 8D	HPAT 1	FILE
8B40-	60 C0 8D 60 C0 8D 60 C0		
8B48-	8D 60 C0 8D 60 C0 8D 60		
8B50-	C0 88 F0 32 D0 D0 B9 00		
8B58-	8C 30 D0 10 00 29 7F AA		
8B60-	9D 00 C0 8D 60 C0 8D 60	HPAT 4	FILE
8B68-	C0 8D 60 C0 8D 60 C0 8D		
8B70-	60 C0 8D 60 C0 8D 60 C0		
8B78-	8D 60 C0 8D 60 C0 8D 60		
8B80-	C0 88 F0 02 D0 D0 4C C2		
8B88-	8C B9 00 8C 30 30 10 00		
8B90-	29 7F AA 9D 00 C0 8D 60	HPAT 2	FILE
8B98-	C0 8D 60 C0 8D 60 C0 8D		
8BA0-	60 C0 8D 60 C0 8D 60 C0		
8BA8-	8D 60 C0 8D 60 C0 8D 60		
8BB0-	C0 8D 60 C0 88 F0 CF D0		
8BB8-	D0 B9 00 8C 30 9D 10 00		
8BC0-	29 7F AA 9D 00 C0 8D 60	HPAT 3	FILE
8BC8-	C0 8D 60 C0 8D 60 C0 8D		
8BD0-	60 C0 8D 60 C0 8D 60 C0		
8BD8-	8D 60 C0 8D 60 C0 8D 60		
8BE0-	C0 8D 60 C0 88 F0 9F D0		
8BE8-	D0 A9 20 20 A8 FC A9 14		
8BF0-	20 A8 FC A9 06 20 A8 FC		
8BF8-	60 49 20 1D 40 20 52 20		
8C00-	60 60 60 60 60 60 60 60		
8C08-	60 60 60 60 60 60 60 60		
8C10-	60 60 60 60 60 60 60 60		
8C18-	60 60 60 60 60 60 60 60		
8C20-	60 60 60 60 60 60 60 60		
8C28-	60 60 60 60 60 60 60 60		
8C30-	60 60 60 60 60 60 60 60		
8C38-	60 60 60 60 60 60 60 60		
8C40-	60 60 60 60 60 60 60 60		
8C48-	60 60 60 60 60 60 60 60	VPATR	FILE
8C50-	60 60 60 60 60 60 60 60		
8C58-	60 60 60 60 60 60 60 60		
8C60-	60 60 60 60 60 60 60 60		
8C68-	60 60 60 60 60 60 60 60		
8C70-	60 60 60 60 60 60 60 60		
8C78-	60 60 60 60 60 60 60 60		
8C80-	60 60 60 60 60 60 60 60		
8C88-	60 60 60 60 60 60 60 60		
8C90-	60 60 60 60 60 60 60 60		
8C98-	60 60 60 60 60 60 60 60		
8CA0-	60 60 60 60 60 60 60 60		
8CA8-	60 60 60 60 60 60 60 60		
8CB0-	60 60 60 60 60 60 60 60		
8CB8-	60 60 60 60 60 60 60 60		
8CC0-	60 60 20 CB 8C 20 E9 8B		
8CC8-	4C 24 8B 2C FB 8C 10 0B		
8CD0-	2C 00 C0 10 06 68 68 60		
8CD8-	EA EA EA EE FD 8C 2C FB		
8CE0-	8C 50 0F A9 1F 2D FD 8C		
8CE8-	D0 0C CE FC 8C D0 0B 68		
8CF0-	68 60 EA EA EA EA EA EA	CONTROL	FILE
8CF8-	EA EA 60 C4 00 00 AD FB		
8D00-	8C 29 3F 8D FC 8C 60		

Fig. 5-5. Hex dumps of VFFS files used in

TWO DEMOS

Program 5-2 is a HIRES demo program that loads several different VFFS subroutines for you. These subroutines are called VFFS.BOXES, VFFS.GRAPH, VFFS.GIRLS, and VFFS.BYE. You'll find hex dumps of these shown in Fig. 5-5. Once again, all of these custom field mixers are created by starting with VFFS.EMPTY, listing it in machine language, changing a few locations using your pattern worksheets, and, then, saving the result to disk under a new name.

The demo program also shows us how we can mix action during normal display times with stunning results during mixed field times. We've purposely slowed things down with SPEED commands and lengthy calculations to make the demo more interesting to watch.

Our demo first mixes text with color LORES boxes and, then, for an encore, puts a pair of HIRES boxes around the whole works. Actually, we stay in a 3-way HIRES-LORES-text mix all along. The HIRES parts are black at first.

We then draw a graph that uses HIRES for the axis and curves, and which uses text for the vertical and horizontal data values. Above that is a text title. Note

VFFS. GRAPH	
8AFF- EA	
8B00- 20 FE BC 2C 60 C0 10 FB	
8B08- 2C 60 C0 30 FB EA 10 00	
8B10- A9 3B 20 A8 FC A9 34 20	
8B18- A8 FC A9 01 20 A8 FC 2C	
8B20- 60 C0 10 EC A0 C0 B9 00	
8B28- 8C 30 63 10 00 29 7F AA	
8B30- 9D 00 C0 8D 51 C0 8D 57	
8B38- C0 8D 54 C0 8D 60 C0 8D	
8B40- 60 C0 8D 60 C0 8D 60 C0	
8B48- 8D 60 C0 8D 60 C0 8D 60	
8B50- C0 88 F0 32 D0 D0 B9 00	
8B58- 8C 30 D0 10 00 29 7F AA	
8B60- 9D 00 C0 8D 51 C0 8D 60	
8B68- C0 8D 50 C0 8D 60 C0 8D	
8B70- 60 C0 8D 60 C0 8D 60 C0	
8B78- 8D 60 C0 8D 60 C0 8D 60	
8B80- C0 88 F0 02 D0 D0 4C C2	
8B88- 8C B9 00 8C 30 30 10 00	
8B90- 29 7F AA 9D 00 C0 8D 51	
8B98- C0 8D 60 C0 8D 50 C0 8D	
8BA0- 60 C0 8D 60 C0 8D 60 C0	
8BA8- 8D 60 C0 8D 60 C0 8D 60	
8BB0- C0 8D 60 C0 88 F0 CF D0	
8BB8- D0 B9 00 8C 30 9D 10 00	
8BC0- 29 7F AA 9D 00 C0 8D 51	
8BC8- C0 8D 60 C0 8D 50 C0 8D	
8BD0- 60 C0 8D 60 C0 8D 60 C0	
8BD8- 8D 60 C0 8D 60 C0 8D 51	
8BE0- C0 8D 60 C0 88 F0 9F D0	
8BE8- D0 A9 20 20 A8 FC A9 14	
8BF0- 20 A8 FC A9 06 20 A8 FC	
8BF8- 60 20 20 20 20 20 20 20	
8C00- 51 51 51 51 51 51 51 51	
8C08- 51 51 51 51 51 51 51 51	
8C10- 51 51 51 51 51 51 51 51	
8C18- D1 51 51 51 51 51 51 51	
8C20- 51 51 51 51 51 51 51 51	
8C28- 51 51 51 51 51 51 51 51	
8C30- 51 51 51 51 51 51 51 51	
8C38- 51 51 51 51 51 51 51 51	
8C40- 51 51 51 51 51 51 51 D1	
8C48- 51 51 51 51 51 51 51 51	
8C50- 51 51 51 51 51 51 51 51	
8C58- 51 51 51 51 51 51 51 51	
8C60- 51 51 51 51 51 51 51 51	
8C68- 51 51 51 51 51 51 51 51	
8C70- 51 51 51 51 51 51 51 51	
8C78- 51 51 51 51 51 51 51 51	
8C80- 51 51 51 51 51 51 51 51	
8C88- 51 51 51 51 51 51 51 51	
8C90- 51 51 51 51 51 51 51 51	
8C98- 51 51 51 51 51 51 51 51	
8CA0- D1 51 51 51 51 51 51 51	
8CA8- D1 51 51 51 51 51 51 51	
8CB0- 51 51 51 51 51 51 51 51	
8CB8- 51 51 51 51 51 51 51 51	
8CC0- 51 60 20 CB 8C 20 E9 8B	
8CC8- 4C 24 8B 2C FB 8C 10 08	
8CD0- 2C 00 C0 10 06 68 68 60	
8CD8- EA EA EA EE FD 8C 2C FB	
8CE0- 8C 50 0F A9 1F 2D FD 8C	
8CE8- D0 0C CE FC 8C D0 0B 68	
8CF0- 68 60 EA EA EA EA EA EA	
8CF8- EA EA 60 80 00 80 AD FB	
8D00- 8C 29 3F 8D FC 8C 60	

VFFS. GIRLS	
8AFF- EA	
8B00- 20 FE BC 2C 60 C0 10 FB	
8B08- 2C 60 C0 30 FB EA 10 00	
8B10- A9 3B 20 A8 FC A9 34 20	
8B18- A8 FC A9 01 20 A8 FC 2C	
8B20- 60 C0 10 EC A0 C0 B9 00	
8B28- 8C 30 63 10 00 29 7F AA	
8B30- 9D 00 C0 8D 52 C0 8D 60	
8B38- C0 8D 60 C0 8D 60 C0 8D	
8B40- 60 C0 8D 60 C0 8D 60 C0	
8B48- 8D 60 C0 8D 60 C0 8D 60	
8B50- C0 88 F0 32 D0 D0 B9 00	
8B58- 8C 30 D0 10 00 29 7F AA	
8B60- 9D 00 C0 8D 60 C0 8D 60	
8B68- C0 8D 60 C0 8D 60 C0 8D	
8B70- 60 C0 8D 60 C0 8D 60 C0	
8B78- 8D 60 C0 8D 60 C0 8D 60	
8B80- C0 88 F0 02 D0 D0 4C C2	
8B88- 8C B9 00 8C 30 30 10 00	
8B90- 29 7F AA 9D 00 C0 8D 60	
8B98- C0 8D 60 C0 8D 60 C0 8D	
8BA0- 60 C0 8D 60 C0 8D 60 C0	
8BA8- 8D 60 C0 8D 60 C0 8D 60	
8BB0- C0 8D 60 C0 88 F0 CF D0	
8BB8- D0 B9 00 8C 30 9D 10 00	
8BC0- 29 7F AA 9D 00 C0 8D 60	
8BC8- C0 8D 60 C0 8D 60 C0 8D	
8BD0- 60 C0 8D 60 C0 8D 60 C0	
8BD8- 8D 60 C0 8D 60 C0 8D 60	
8BE0- C0 8D 60 C0 88 F0 9F D0	
8BE8- D0 A9 20 20 A8 FC A9 14	
8BF0- 20 A8 FC A9 06 20 A8 FC	
8BF8- 60 49 20 1D 40 20 52 20	
8C00- 56 56 60 60 60 60 60 60	
8C08- 51 60 60 60 60 60 60 60	
8C10- 57 60 60 60 60 60 60 60	
8C18- 50 57 60 60 60 60 60 60	
8C20- 51 60 60 60 60 60 60 60	
8C28- 50 60 60 60 60 60 60 60	
8C30- 51 60 60 60 60 60 60 60	
8C38- 50 60 60 60 60 60 60 60	
8C40- 51 60 60 60 60 60 60 60	
8C48- 50 60 60 60 60 60 60 60	
8C50- 51 60 60 60 60 60 60 60	
8C58- 50 60 60 60 60 60 60 60	
8C60- 51 60 60 60 60 60 60 60	
8C68- 50 60 60 60 60 60 60 60	
8C70- 51 60 60 60 60 60 60 60	
8C78- 50 60 60 60 60 60 60 60	
8C80- 51 60 60 60 60 60 60 60	
8C88- 50 60 60 60 60 60 60 60	
8C90- 51 60 60 60 60 60 60 60	
8C98- 50 60 60 60 60 60 60 60	
8CA0- 51 60 60 60 60 60 60 60	
8CA8- 50 60 60 60 60 60 60 60	
8CB0- 51 60 60 60 60 60 60 60	
8CB8- 50 60 60 60 60 60 60 60	
8CC0- 51 60 20 CB 8C 20 E9 8B	
8CC8- 4C 24 8B 2C FB 8C 10 08	
8CD0- 2C 00 C0 10 06 68 68 60	
8CD8- EA EA EA EE FD 8C 2C FB	
8CE0- 8C 50 0F A9 1F 2D FD 8C	
8CE8- D0 0C CE FC 8C D0 0B 68	
8CF0- 68 60 EA EA EA EA EA EA	
8CF8- EA EA 60 80 00 A5 AD FB	
8D00- 8C 29 3F 8D FC 8C 60	

VFFS. BYE	
8AFF- EA	
8B00- 20 FE BC 2C 60 C0 10 FB	
8B08- 2C 60 C0 30 FB EA 10 00	
8B10- A9 3B 20 A8 FC A9 34 20	
8B18- A8 FC A9 01 20 A8 FC 2C	
8B20- 60 C0 10 EC A0 C0 B9 00	
8B28- 8C 30 63 10 00 29 7F AA	
8B30- 9D 00 C0 8D 60 C0 8D 60	
8B38- C0 8D 60 C0 8D 60 C0 8D	
8B40- 60 C0 8D 60 C0 8D 60 C0	
8B48- 8D 60 C0 8D 60 C0 8D 60	
8B50- C0 88 F0 32 D0 D0 B9 00	
8B58- 8C 30 D0 10 00 29 7F AA	
8B60- 9D 00 C0 8D 60 C0 8D 60	
8B68- C0 8D 60 C0 8D 60 C0 8D	
8B70- 60 C0 8D 60 C0 8D 60 C0	
8B78- 8D 60 C0 8D 60 C0 8D 60	
8B80- C0 88 F0 02 D0 D0 4C C2	
8B88- 8C B9 00 8C 30 30 10 00	
8B90- 29 7F AA 9D 00 C0 8D 60	
8B98- C0 8D 60 C0 8D 60 C0 8D	
8BA0- 60 C0 8D 60 C0 8D 60 C0	
8BA8- 8D 60 C0 8D 60 C0 8D 60	
8BB0- C0 8D 60 C0 88 F0 CF D0	
8BB8- D0 B9 00 8C 30 9D 10 00	
8BC0- 29 7F AA 9D 00 C0 8D 60	
8BC8- C0 8D 60 C0 8D 60 C0 8D	
8BD0- 60 C0 8D 60 C0 8D 60 C0	
8BD8- 8D 60 C0 8D 60 C0 8D 60	
8BE0- C0 8D 60 C0 88 F0 9F D0	
8BE8- D0 A9 20 20 A8 FC A9 14	
8BF0- 20 A8 FC A9 06 20 A8 FC	
8BF8- 60 49 20 1D 40 20 52 20	
8C00- 51 60 60 60 60 60 60 60	
8C08- 50 60 60 60 60 60 60 60	
8C10- 60 60 60 60 60 60 60 60	
8C18- 60 60 60 60 60 60 60 60	
8C20- 60 60 60 60 60 60 60 60	
8C28- 60 60 60 60 60 60 60 60	
8C30- 60 60 60 60 60 60 60 60	
8C38- 60 60 60 60 60 60 60 60	
8C40- 60 60 60 60 60 60 60 60	
8C48- 60 60 60 60 60 60 60 60	
8C50- 60 60 60 60 60 60 60 60	
8C58- 60 60 60 60 60 60 60 60	
8C60- 56 60 60 60 60 60 60 60	
8C68- 60 60 60 60 60 60 60 60	
8C70- 60 60 60 60 60 60 60 60	
8C78- 60 60 60 60 60 60 60 60	
8C80- 50 57 60 60 60 60 60 60	
8C88- 60 60 60 60 60 60 60 60	
8C90- 60 60 60 60 60 60 60 60	
8C98- 60 60 60 60 60 60 60 60	
8CA0- 60 60 60 60 60 60 60 60	
8CA8- 60 60 60 60 60 60 60 60	
8CB0- 60 60 60 60 60 60 60 60	
8CB8- 60 60 60 60 60 60 60 60	
8CC0- 51 60 20 CB 8C 20 E9 8B	
8CC8- 4C 24 8B 2C FB 8C 10 08	
8CD0- 2C 00 C0 10 06 68 68 60	
8CD8- EA EA EA EE FD 8C 2C FB	
8CE0- 8C 50 0F A9 1F 2D FD 8C	
8CE8- D0 0C CE FC 8C D0 0B 68	
8CF0- 68 60 EA EA EA EA EA EA	
8CF8- EA EA 60 80 00 80 AD FB	
8D00- 8C 29 3F 8D FC 8C 60	

"Fun With Mixed Fields" demo programs.

the use of a vertical HIRES line in order to get rid of an on-screen glitch in line 2220.

After our graph is complete, we then inset some flashing text inside the HIRES display. It is usually very tricky to have flashing text in the middle of a HIRES screen. Mixed fields make it trivial.

The subject of our next plot is obvious. We show how to do a LORES horizontal bar graph with text documentation. A HIRES graph axis completes the picture for us.

The final display in the Fun With Mixed Field demonstration program shows how you can have a LORES, HIRES, and text words on the screen at the same time right above each other.

A totally different kind of Applesoft demo appears as Program 5-3. This one is called LORES COLORS 121 and shows 121 of the many available LORES colors. We purposely aren't going to tell you how this one works. But, if you combine the detective work you should have picked up in Enhancement 3 with a good understanding of your VPATRN and HPAT files from this enhancement,

VFFS. LORES	LORES 1 (DISPLAY PAGE 1)
8AFF- EA	0400- 11 11 00 11 11 00 11 11
8B00- 20 FE 8C 2C 60 C0 10 FB	0408- 00 11 11 00 11 11 00 11
8B08- 2C 60 C0 30 FB EA 10 00	0410- 11 00 00 00 00 11 11 00
8B10- A9 3B 20 A8 FC A9 34 20	0418- 11 11 00 11 11 00 11 11
8B18- A8 FC A9 01 20 A8 FC 2C	0420- 00 11 11 00 11 11 00 00
8B20- 60 C0 10 EC A0 C0 B9 00	0428- 44 44 00 44 44 00 44 44
8B28- 8C 30 63 10 00 29 7F AA	0430- 00 44 44 00 44 44 00 44
8B30- 9D 00 C0 8D 50 C0 8D 56	0438- 44 00 30 30 00 55 55 00
8B38- C0 8D 60 C0 8D 60 C0 8D	0440- 55 55 00 55 55 00 55 55
8B40- 60 C0 8D 60 C0 8D 60 C0	0448- 00 55 55 00 55 55 00 00
8B48- 8D 60 C0 8D 60 C0 8D 60	0450- 66 66 00 66 66 00 66 66
8B50- C0 88 F0 32 D0 D0 B9 00	0458- 00 77 77 00 77 77 00 77
8B58- 8C 30 D0 10 00 29 7F AA	0460- 77 00 00 00 77 77 00 77
8B60- 9D 00 C0 8D 60 C0 8D 60	0468- 77 77 00 77 77 00 77 77
8B68- C0 8D 60 C0 8D 60 C0 8D	0470- 00 77 77 00 77 77 00 00
8B70- 60 C0 8D 60 C0 8D 60 C0	0478- 12 00 37 37 37 37 24 37
8B78- 8D 60 C0 8D 60 C0 8D 60	0480- 01 01 00 01 01 00 01 01
8B80- C0 88 F0 02 D0 D0 4C C2	0488- 00 01 01 00 01 01 00 01
8B88- 8C B9 00 8C 30 10 10 00	0490- 01 00 00 00 01 01 00 00
8B90- 29 7F AA 9D 00 C0 8D 60	0498- 01 01 00 01 01 00 01 01
8B98- C0 8D 60 C0 8D 51 C0 8D	04A0- 00 01 01 00 01 01 00 00
8BA0- 60 C0 8D 60 C0 8D 60 C0	04A8- 04 04 00 04 04 00 04 04
8BA8- 8D 60 C0 8D 50 C0 8D 60	04B0- 00 04 04 00 04 04 00 04
8BB0- C0 8D 60 C0 88 F0 CF D0	04B8- 04 00 33 33 00 05 05 00
8BB8- D0 B9 00 8C 30 9D 10 00	04C0- 05 05 00 05 05 00 05 05
8BC0- 29 7F AA 9D 00 C0 8D 60	04C8- 00 05 05 00 05 05 00 00
8BC8- C0 8D 60 C0 8D 60 C0 8D	04D0- 80 80 00 80 80 00 80 80
8BD0- 60 C0 8D 60 C0 8D 60 C0	04D8- 00 80 80 00 80 80 00 80
8BD8- 8D 60 C0 8D 60 C0 8D 60	04E0- 80 00 00 00 80 80 00 00
8BE0- C0 8D 60 C0 88 F0 9F D0	04E8- 80 80 00 90 90 00 90 90
8BE8- D0 A9 20 20 A8 FC A9 14	04F0- 00 90 90 00 90 90 00 00
8BF0- 20 A8 FC A9 06 20 A8 FC	04F8- 04 37 37 37 37 37 00 37
8BF8- 60 20 20 20 20 20 20 20	
8C00- 54 55 54 55 54 55 54 55	0500- 11 11 00 11 11 00 11 11
8C08- 54 55 54 55 54 55 54 55	0508- 00 22 22 00 22 22 00 22
8C10- 54 55 54 55 54 55 54 55	0510- 22 00 00 00 00 22 22 00
8C18- 54 55 54 55 54 55 54 55	0518- 22 22 00 22 22 00 22 22
8C20- 54 55 54 55 54 55 54 55	0520- 00 22 22 00 22 22 00 00
8C28- 54 55 54 55 54 55 54 55	0528- 00 00 00 00 00 00 00 33
8C30- 54 55 54 55 54 55 54 55	0530- 03 03 03 03 03 03 03 03
8C38- 54 55 54 55 54 55 54 55	0538- 03 03 03 03 03 03 03 03
8C40- 54 55 54 55 54 55 54 55	0540- 03 03 03 03 03 03 33 00
8C48- 54 55 54 55 54 55 54 55	0548- 00 00 00 00 00 00 00 00
8C50- 54 55 54 55 54 55 54 55	0550- 88 88 00 88 88 00 88 88
8C58- 60 60 60 60 60 60 60 60	0558- 00 88 88 00 88 88 00 88
8C60- E0 60 60 60 60 60 60 60	0560- 88 00 00 00 88 88 00 00
8C68- E0 60 60 60 60 60 60 60	0568- 88 88 00 99 99 00 99 99
8C70- 60 60 60 60 54 55 54 55	0570- 00 99 99 00 99 99 00 00
8C78- 54 55 54 55 54 55 54 55	0578- 2F 80 37 37 37 37 37 37
8C80- 54 55 54 55 54 55 54 55	0580- 01 01 00 01 01 00 01 01
8C88- 54 55 54 55 54 55 54 55	0588- 00 02 02 00 02 02 00 02
8C90- 54 55 54 55 54 55 54 55	0590- 02 00 00 00 00 02 02 00
8C98- 54 55 54 55 54 55 54 55	0598- 02 02 00 02 02 00 02 02
8CA0- 54 55 54 55 54 55 54 55	05A0- 00 02 02 00 02 02 00 00
8CA8- 54 55 54 55 54 55 54 55	05A8- 00 00 00 00 00 00 00 33
8CB0- 54 55 54 55 54 55 54 55	05B0- A0 A0 A0 B1 B2 B1 A0 A0
8CB8- 54 55 54 55 54 55 54 55	05B8- CC CF D2 C5 D3 A0 C3 CF
8CC0- 54 55 20 CB 8C 20 E9 8B	05C0- CC CF D2 D3 00 00 33 00
8CC8- 4C 24 8B 2C FB 8C 10 08	05C8- 00 00 00 00 00 00 00 00
8CD0- 2C 00 C0 10 06 68 68 60	05D0- 90 90 00 90 90 00 90 90
8CD8- EA EA EA EE FD 8C 2C FB	05D8- 00 A0 A0 00 A0 A0 00 A0
8CE0- 8C 50 0F A9 1F 2D FD 8C	05E0- A0 00 00 00 00 A0 A0 00
8CE8- D0 0C CE FC 8C D0 0B 68	05E8- A0 A0 00 A0 A0 00 B0 B0
8CF0- 68 60 EA EA EA EA EA EA	05F0- 00 B0 B0 00 B0 B0 00 00
8CF8- EA EA 60 80 00 A5 AD FB	05F8- 60 03 37 37 37 37 37 37
8D00- 8C 29 3F 8D FC 8C 60	
	0600- 22 22 00 22 22 00 22 22
	0608- 00 22 22 00 22 22 00 33
	0610- 33 00 00 00 00 33 33 00
	0618- 33 33 00 33 32 00 33 33
	0620- 00 33 33 00 33 33 00 00
	0628- 00 00 00 00 00 00 00 33
	0630- 30 30 30 30 30 30 30 30
	0638- 30 30 30 30 30 30 30 30
	0640- 30 30 30 30 30 30 33 00
	0648- 00 00 00 00 00 00 00 00
	0650- 99 99 00 99 99 00 99 99
	0658- 00 AA AA 00 AA AA 00 AA
	0660- AA 00 00 00 00 AA AA 00
	0668- AA AA 00 AA AA 00 BB BB
	0670- 00 BB BB 00 BB BB 00 00
	0678- 60 09 3F 3F 3F 3F 3F 3F
	0680- 02 02 00 02 02 00 02 02
	0688- 00 02 02 00 02 02 00 03
	0690- 03 00 00 00 00 03 03 00
	0698- 03 03 00 03 03 00 03 03
	06A0- 00 03 03 00 03 03 00 00
	06A8- 50 50 00 50 50 00 50 50
	06B0- 00 50 50 00 50 50 00 60
	06B8- 60 00 33 33 00 60 60 00
	06C0- 60 60 00 60 60 00 60 60
	06C8- 00 60 60 00 60 60 00 00
	06D0- 80 80 00 80 80 00 C0 C0
	06D8- 00 C0 C0 00 C0 C0 00 C0
	06E0- 00 00 00 00 00 00 00 00
	06E8- D0 D0 00 D0 D0 00 E0 E0
	06F0- 00 E0 E0 00 F0 F0 00 00
	06F8- 02 3F 3F 3F 3F 3F 3F 3F

Note: These values must be indirectly loaded since they are an "IMAGE" of text page 1.

Fig. 5-6. Hex dumps of VFFS and SCREEN files

you will be well on your way to thoroughly understanding some machine language secrets.

Fig. 5-6 shows us hex dumps of the VFFS.LORES subroutine, along with two display page files called LORES1 and LORES2. The companion diskette to this book has these programs ready to go, and also includes two programs called LORES1 CREATE and LORES2 CREATE. You can modify these CREATE programs for other LORES color demonstrations. Do not run either CREATE program with a locked LORES1 or LORES2 on the same disk or you will get an error message.

Note that you cannot directly "hand load" LORES1 since this is on the display page. Instead, you save a version from \$0800 to \$0BFF onto your disk and, then, read this VERSION into \$0400 to \$07FF. The demo disk does all this for you.

Oh, yes. One gotcha. Before you can use an Applesoft program such as LORES COLORS 121 and page Two LORES or text together, you have to make sure the Applesoft program starts above memory location \$0C00. The copy of Program 5-3 that is on the disk has an automatic repositioner built in. The disk

LORES 2 (DISPLAY PAGE 2)

0800- 11 11 00 22 22 00 33 33	0A00- BB BB 00 CC CC 00 DD DD
0808- 00 44 44 00 55 55 00 66	0A08- 00 EE EE 00 FF FF 00 33
0810- 66 00 00 00 00 77 77 00	0A10- 33 00 00 00 00 44 44 00
0818- 88 88 00 99 99 00 AA AA	0A18- 55 55 00 66 66 00 77 77
0820- 00 BB BB 00 CC CC 00 00	0A20- 00 88 88 00 99 99 00 00
0828- AA AA 00 BB BB 00 CC CC	0A28- 00 00 00 00 00 00 00 00
0830- 00 DD DD 00 EE EE 00 FF	0A30- 00 00 00 00 00 00 00 00
0838- FF 00 00 00 00 55 55 00	0A38- 00 00 00 00 00 00 00 00
0840- 66 66 00 77 77 00 88 88	0A40- 00 00 00 00 00 00 00 00
0848- 00 99 99 00 AA AA 00 00	0A48- 00 00 00 00 00 00 00 00
0850- DD DD 00 EE EE 00 FF FF	0A50- DD DD 00 EE EE 00 FF FF
0858- 00 77 77 00 88 88 00 99	0A58- 00 AA AA 00 BB BB 00 CC
0860- 99 00 00 00 00 AA AA 00	0A60- CC 00 00 00 00 DD DD 00
0868- BB BB 00 CC CC 00 DD DD	0A68- EE EE 00 FF FF 00 BB BB
0870- 00 EE EE 00 FF FF 00 00	0A70- 00 CC CC 00 DD DD 00 00
0878- 13 00 37 37 37 37 26 37	0A78- 60 07 3F 3F 3F 3F 3F 3F
0880- 01 01 00 02 02 00 03 03	0A80- 0B 0B 00 0C 0C 00 0D 0D
0888- 00 04 04 00 05 05 00 06	0A88- 00 0E 0E 00 0F 0F 00 03
0890- 06 00 00 00 00 07 07 00	0A90- 03 00 00 00 00 04 04 00
0898- 08 08 00 09 09 00 0A 0A	0A98- 05 05 00 06 06 00 07 07
08A0- 00 0B 0B 00 0C 0C 00 00	0AA0- 00 0B 0B 00 09 09 00 00
08A8- 0A 0A 00 0B 0B 00 0C 0C	0AA8- 80 80 00 00 00 00 00 00
08B0- 00 0D 0D 00 0E 0E 00 0F	0AB0- 00 E0 E0 00 F0 F0 00 60
08B8- 0F 00 00 00 00 05 05 00	0AB8- 60 00 00 00 00 70 70 00
08C0- 06 06 00 07 07 00 08 08	0AC0- 80 80 00 90 90 00 A0 A0
08C8- 00 09 09 00 0A 0A 00 00	0AC8- 00 B0 B0 00 C0 C0 00 00
08D0- 80 80 00 90 90 00 A0 A0	0AD0- E0 E0 00 F0 F0 00 C0 C0
08D8- 00 B0 B0 00 C0 C0 00 00	0AD8- 00 D0 D0 00 E0 E0 00 F0
08E0- D0 D0 00 00 00 E0 E0 00	0AE0- F0 00 00 00 00 D0 D0 00
08E8- F0 F0 00 90 90 00 A0 A0	0AE8- E0 E0 00 F0 F0 00 E0 E0
08F0- 00 B0 B0 00 C0 C0 00 00	0AF0- 00 F0 F0 00 F0 F0 00 00
08F8- 04 37 37 37 37 37 00 37	0AF8- 02 3F 3F 3F 3F 3F 3F 3F
0900- DD DD 00 EE EE 00 FF FF	0B00- AA AA 00 BB BB 00 CC CC
0908- 00 22 22 00 33 33 00 44	0B08- 00 DD DD 00 EE EE 00 FF
0910- 44 00 00 00 00 55 55 00	0B10- FF 00 00 00 00 44 44 00
0918- 66 66 00 77 77 00 88 88	0B18- 55 55 00 66 66 00 77 77
0920- 00 99 99 00 AA AA 00 00	0B20- 00 88 88 00 99 99 00 00
0928- 00 00 00 00 00 00 00 00	0B28- BB BB 00 CC CC 00 DD DD
0930- 00 00 00 00 00 00 00 00	0B30- 00 EE EE 00 FF FF 00 66
0938- 00 00 00 00 00 00 00 00	0B38- 66 00 00 00 00 77 77 00
0940- 00 00 00 00 00 00 00 00	0B40- 88 88 00 99 99 00 AA AA
0948- 00 00 00 00 00 00 00 00	0B48- 00 BB BB 00 CC CC 00 00
0950- 88 88 00 99 99 00 AA AA	0B50- EE EE 00 FF FF 00 CC CC
0958- 00 BB BB 00 CC CC 00 DD	0B58- 00 DD DD 00 EE EE 00 FF
0960- DD 00 00 00 00 EE EE 00	0B60- FF 00 00 00 00 DD DD 00
0968- FF FF 00 99 99 00 AA AA	0B68- EE EE 00 FF FF 00 EE EE
0970- 00 BB BB 00 CC CC 00 00	0B70- 00 FF FF 00 FF FF 00 00
0978- 30 00 37 37 37 37 37 37	0B78- 3F FF 3F 3F 3F 3F 3F 3F
0980- 0D 0D 00 0E 0E 00 0F 0F	0B80- 0A 0A 00 0B 0B 00 0C 0C
0988- 00 02 02 00 03 03 00 04	0B88- 00 0D 0D 00 0E 0E 00 0F
0990- 04 00 00 00 00 05 05 00	0B90- 0F 00 00 00 00 04 04 00
0998- 06 06 00 07 07 00 08 08	0B98- 05 05 00 06 06 00 07 07
09A0- 00 09 09 00 0A 0A 00 00	0BA0- 00 0B 0B 00 09 09 00 00
09A8- 00 00 00 00 00 00 00 00	0BA8- D0 D0 00 E0 E0 00 F0 F0
09B0- 00 00 00 00 00 00 00 00	0BB0- 00 70 70 00 80 80 00 90
09B8- 00 00 00 00 00 00 00 00	0BB8- 90 00 00 00 00 A0 A0 00
09C0- 00 00 00 00 00 00 00 00	0BC0- B0 B0 00 C0 C0 00 D0 D0
09C8- 00 00 00 00 00 00 00 00	0BC8- 00 E0 E0 00 F0 F0 00 00
09D0- D0 D0 00 E0 E0 00 F0 F0	0BD0- 00 00 00 00 00 00 00 00
09D8- 00 A0 A0 00 B0 B0 00 C0	0BD8- 00 00 00 00 00 00 00 00
09E0- C0 00 00 00 00 D0 D0 00	0BE0- 00 00 00 00 00 00 00 00
09E8- E0 E0 00 F0 F0 00 B0 B0	0BE8- 00 00 00 00 00 00 00 00
09F0- 00 C0 C0 00 D0 D0 00 00	0BF0- 00 00 00 00 00 00 00 00
09F8- 60 03 37 37 37 37 37 37	0BF8- C6 3F 3F 3F 3F 3F 3F 3F

Note: Page two text must be protected to use this listing.

used in the LORES COLOR 121 demonstration.

version of LORES COLORS 121 will run just like any ordinary program. But, if you are trying to copy the LORES COLORS 121 program from this book, do a POKE 104,12 and a POKE 3072,0 from the keyboard *before* entering the program and, again, immediately *before* every use. Always turn your Apple off and back on again before running anything else on the machine. We might look at repositioning details in a future enhancement.

So far, I have been able to find only 136 LORES colors. Sorry about that. I left the ugliest 15 colors off the display on purpose. But, even these uglies might be useful in order to add texture to a black and white display. How many new LORES colors can you find?

There are lots of possible VFFS options and improvements since mixed fields is a brand new ball game. You might like to write a program that will automatically generate custom VFFS files for you. You might like to modify VFFS.EMPTY to put a "phase shifter" between the exact lock and the rest of the program. This lets you switch on characters 0,4,8, . . . , or characters 1, 5, 9, . . . , or characters 2, 6, 10, . . . , or characters 3,7,11, . . . , per your choosing. The really good

PROGRAM 5-3
LORES COLORS 121

LANGUAGE: APPLESOFT

NEEDS: FIELD SYNC MOD
VFFS. LORES
RELOCATION ABOVE TEXT 2
LORES 1
LORES 2

```
10 REM
12 REM *****
14 REM *
16 REM * LORES COLORS 121 *
18 REM *
20 REM * COPYRIGHT 1981 *
22 REM * BY DON LANCASTER *
24 REM * AND SYNERGETICS *
26 REM *
28 REM * VERSION 1.0 *
30 REM * ( 9-20-81) *
32 REM * ALL COMMERCIAL *
34 REM * RIGHTS RESERVED *
36 REM *
38 REM *****

50 REM THIS PROGRAM NEEDS THE
52 REM FIELD SYNC MODIFICATION
54 REM AND BINARY FILES LORES1,
56 REM LORES2, AND VFFS.LORES
58 REM TO WORK PROPERLY.

70 REM SEE ENHANCING YOUR
72 REM APPLE II, VOLUME 1
74 REM FOR MORE USE DETAILS.

80 REM WARNING: THIS PROGRAM
82 REM MUST BE RELOCATED
84 REM ABOVE TEXT PAGE TWO!

86 REM DO A POKE 3072,0:
88 REM AND A POKE 104,12
90 REM BEFORE USING. THIS
92 REM IS DONE AUTOMATICALLY
94 REM IN THE VERSION OF THIS
96 REM PROGRAM ON THE DEMO
98 REM DISK.
```

PROGRAM 5-3, CONT'D...

```
100 HOME : GR : POKE - 16302,0:
    CALL - 1998: REM CLEAR FULL GRAPHICS SCREEN

200 PRINT "BLOAD VFFS.LORES": REM
    CTRL D
210 PRINT "BLOAD LORES1": REM
    CTRL D
220 PRINT "BLOAD LORES2,A$800": REM
    CTRL D
300 CALL - 29952: REM MIX FIELDS

400 POKE - 16300,0: TEXT : HOME
    :
405 POKE 2048,0: POKE 104,8: PRINT
    : PRINT "RUN MENU"
406 REM DELETE 405 IF AUTO MENU
    IS NOT IN USE

410 PRINT "PLEASE TURN APPLE POWER OFF AND BACK ON AGAIN BEFORE RUNNING ANY OTHER PROGRAM."
420 PRINT : PRINT : PRINT "! HASTA LA BYE BYE": VTAB 23: END
```

stuff will happen when you start flipping nonobvious soft switches, for things like external 3-D displays, anti-aliased grey scales, and so on. And, the opportunities for dynamic “change-while-it’s-running” field-switch animation and wipes are awesome.

By the way, if you try flipping the speaker or cassette soft switches out with VFFS, you’ll have to replace any involved HPAT “8D” absolute stores with “AD” absolute loads instead. Due to an Apple quirk, stores whap each soft switch *twice*. This puts the speaker cone right back where it was two microseconds earlier and it produces no sound.

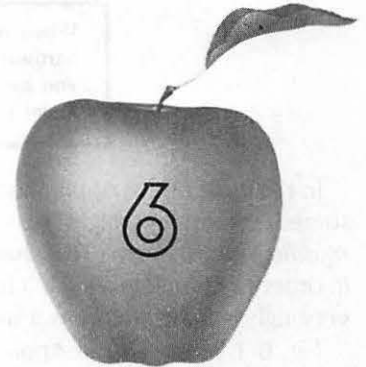
Now, it’s your turn. What can you do with mixed fields? In the back of this volume is a postcard. Use it to show us the best use of mixed fields that you can think of. Pay particular attention to flipping nonobvious switches like the annunciators and whatever. We’ll work up the best of the best into future enhancements that everyone can share 🍏

The following programs and files
are included on the companion
diskette to this volume:

VFFS.EMPTY.SOURCE
VFFS.EMPTY
VFFS.BOXES
VFFS.GRAPH
VFFS.GIRLS
VFFS.BYE
VFFS.LORES
FUN WITH MIXED FIELDS
LORES COLORS 121
LORES1 CREATE
LORES2 CREATE
LORES1
LORES2

All are fully copyable.

Enhancement



GLITCH STOMPER

This simple, three dollar, add-on hardware mod makes the mixed field displays of Enhancement 5 even more powerful. The mod lets you glitchlessly switch to and from LORES anywhere on the screen.

GLITCH STOMPER

As we promised you in Enhancement 5, here is a fairly simple mixed fields add-on hardware mod that will eliminate the worst of the on-screen glitches as you switch to or from LORES on any live portion of your screen. The *glitch stomper* works on any Apple—from Revision 1 on. It costs under \$3.00 and is fairly easy to install or remove. It is only needed for mid-screen LORES field switching.

up the extra HIRES colors during Revision 1. Their idea was to make a few changes as simply and cheaply as possible to the existing Revision 0 Apple board. But, what they forgot, and what we are about to find out is . . .

**With hardware or software, there
NEVER is such a thing as a small
change!**

and even more to the point . . .

When you do make any change in hardware or software, anything you ignore will surely return to haunt you.

In the case of the Apple mod, the thing that the Apple people ignored is that someday someone might like to *instantly* switch to or from LORES while in the *middle* of a live scan. The changes that they made require one character time in order to complete a switch to or from LORES. During that time, you get some very ugly glitches that are a mix of both text and HIRES.

Fig. 6-1 shows us the Apple circuitry that is involved in the modification. A one-of-eight electronic selector was originally provided at location A9 on the Revision 0 Apple boards. This selector switch was set up so that the code on the "A," "B," and "C" select lines picked the source of the video to be routed to the display. When the original switch was in any of positions Zero through Three, you got a text output. This text output came from a 2513 dot matrix character generator by way of a serial video-shift register at A3.

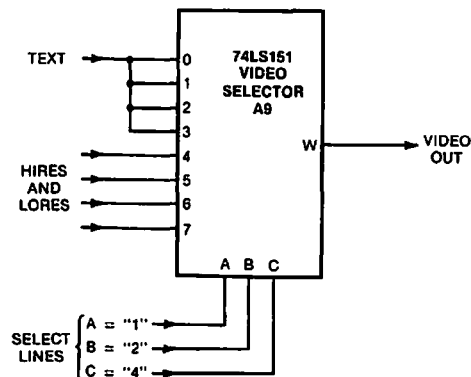


Fig. 6-1. Video selector used in Revision 0 Apple boards. Line "C" unconditionally picks text or graphics.

HIRES and LORES were input to positions Four through Seven of this switch. On Revision 0 boards, the HIRES and LORES were separated in earlier circuits and one or the other, but not both, would go to selector A9. When in a graphics mode, A9 would use its four graphics input positions to complete sorting out the dots on either a HIRES cell or a LORES color block.

Note that the switching between text and graphics here was unconditional. Whenever select line "C" with a weight of "4" was low, you went immediately to the text side of the switch. When select line "C" went high, you went to the graphics side of the switch. Since all four low inputs went to the same text source, it didn't matter what selector line "A" (weighted 1), and selector line "B" (weighted 2), were up to.

Now for the problem. Fig. 6-2 shows the same selector switch after the HIRES color modification, and as used on all Apples of Revision 1 and higher. Now, the bottom two switch positions are text, the next two are HIRES, and the four high ones are LORES, all separate. One of the HIRES inputs is delayed one-quarter of a color cycle from the other, giving you a choice of two sets of colors, depending on whether the most significant bit of the HIRES word was a one or a zero.

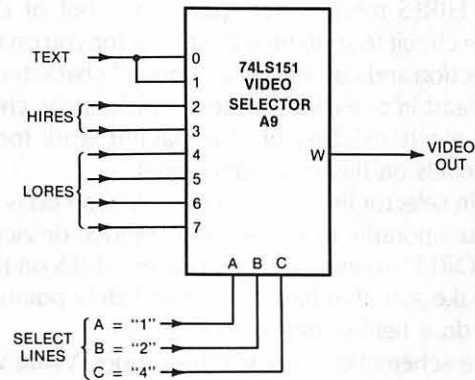


Fig. 6-2. Video selector circuit used in newer Apple boards. Line "C" is no longer unconditional.

Nice and neat.

Except for the problem. To go into text or HIRES, you now have to pay attention to all three select lines, instead of using only select line "C" to unconditionally go to text like before. Apple very carefully worked things out so that the select signals would point exactly to the right switch position. Lines "B" and "C" must work together to unconditionally output text or HIRES.

The only little hassle is that the logic to keep everything pointing in the right place ends up one character *behind* the character you think you are on when you suddenly switch display modes. It takes the Apple one character more to complete the switching to or from LORES and to either text or HIRES.

The result of this is an ugly glitch. Fig. 6-3 shows you what you get. When you suddenly switch out of LORES, you get a symbol that is split into two vertical halves. The top half will be the dot matrix display of the character still stuck in the pipe, while the bottom half will be a HIRES mapping of the actual ASCII code bits of that same "in the pipe" character. Thus, the top half is the character and the bottom half is the code that forms that character.

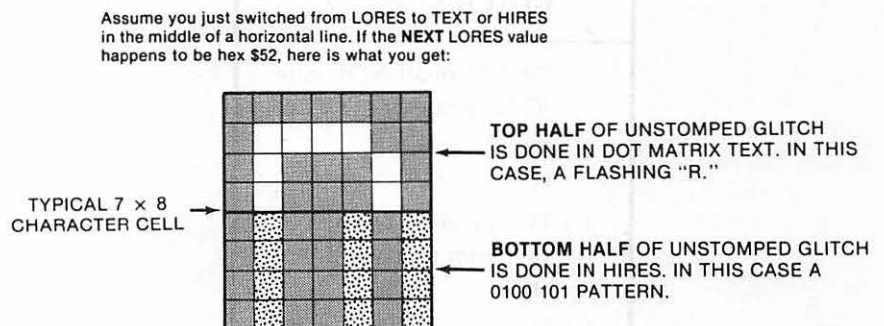


Fig. 6-3. Typical one-character unstomped glitch that you get on a sudden mid-scan switch out of LORES.

The problem to you, as field mixer, is that while you can "hide" either half of this glitch by a choice of what character you put here, you cannot get rid of both halves of the glitch at once. Something always remains to haunt you and foul up the display.

What we have to do is make sure that your selector switch instantly goes into the chosen text or HIRES mode when you switch out of LORES, instead of waiting for the Apple circuit to straighten things out for you on the next character slot. Make this correction and you still get a "wrong" character, but the "wrong" character is now at least in one entire piece. By picking the character code, you can now make this glitch invisible, or else, have it work for you one way or another. We saw details on this in Enhancement 5.

The problem lies in selector line "B." All line "A" can do is pick even or odd, so it can't point us temporarily to text while in HIRES, or vice versa. And, line "C" is what picks LORES on one hand and text or HIRES on the other. So what we have to do is make sure that line "B" immediately points to text or HIRES the instant that we do a field switch out of LORES.

Fig. 6-4 shows the schematic of the glitch stomper. What we do is intercept line "B" of the selector, and force it to a "zero" the instant that we go into a text mode, and to a "one" the instant that we go into a HIRES mode. Should we be in LORES, the signal that is supposed to be on line "B" passes through unharmed.

This speeds up the Apple switching logic by one character, so we can instantly switch to an all-text mode, an all-HIRES mode, or back to LORES anywhere on the screen. Any glitch that remains is in one piece and is easy to handle by changing its code to something useful or invisible.

We will now show you how to make this mod using three new integrated circuits. While you only need one new 14-cent IC to do the job, we are going to chop and channel the other two in a way that the Apple warranty people might find suspect. When the mod is done, two of Apple's integrated circuits will be left over. You can set these aside for use in case you ever need a warranty repair.

As with the hardware mods in the other enhancements, this one does in fact void your Apple warranty. But, if you are careful, you can easily and completely return things back to normal.

Here are the parts you will need . . .

PARTS LIST FOR GLITCH STOMPER	
()	74LS02 quad NOR gate IC (2 needed).
()	74LS151 1-of-8 selector IC.
()	DIP socket, 16-pin, premium machined-pin style.
()	No. 24 insulated solid wire, 4-1/2 inches long, red.
()	No. 24 insulated solid wire, 4-1/2 inches long, blue.
()	Electronic solder (5 inches).

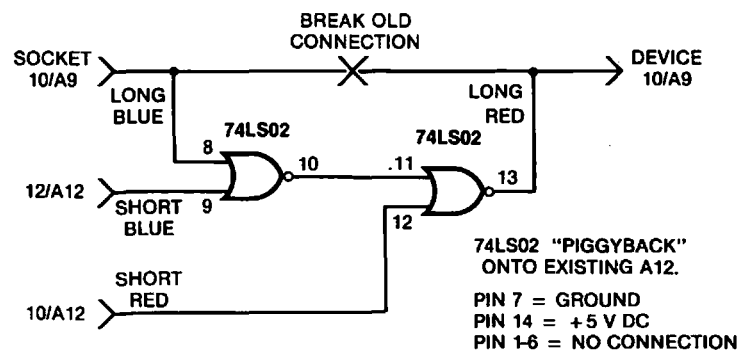


Fig. 6-4. Schematic of the glitch stomper modification.

Here are the tools you will need . . .

TOOLS NEEDED TO BUILD GLITCH STOMPER

- () Small soldering iron, 40 watt
- () Diagonal-cutting pliers
- () Needle nose pliers
- () Wire stripper
- () Piece of protective IC foam
- () IC puller (optional)
- () Phillips screwdriver
- () Regular small screwdriver

Construction details for your glitch stomper are shown in Fig. 6-5. Be sure to use a premium machined-pin DIP socket. This is the only type that can safely be plugged into another socket without damage. Also, be certain you understand how DIP pins are numbered.

Installation is slightly harder than the earlier mods since the case of your Apple will have to be temporarily removed. Here is how to install your glitch stomper . . .

INSTALLING THE GLITCH STOMPER

1. Put a rug or other soft cover over your work area.
2. Turn Apple OFF. Then unplug both ends of the Apple power cord and set the cord aside.

3. Pop the cover on your Apple by pulling sharply up—first at left rear, and then right rear.
4. Remove all plug-in cards, cables, rf modulator leads, and other add-ons. Make a careful record of what goes where.
5. Turn the Apple upside down onto the rug. Using a Phillips screwdriver, remove two screws at extreme rear, two on either extreme side, and four from the front.

NOTE — Remove ONLY these 10 screws. **Do not let the case separate from the rest of the Apple.**

6. Carefully grab *both* the case and the chassis of the Apple and turn them *both* back right-side up together. **Do not let case separate from chassis when you do this.**
7. Gently lift the front of the case only far enough to look inside. Note the keyboard connector. Now, lift the case up and back as far as you can without stressing the keyboard connector.

You should be able to rest the case on the power supply and on a book or two.

8. Verify that there is a 74LS151 integrated circuit in location A9. This is in the front row, somewhat right of center.

9. Remove the 74LS151 at A9 and set it aside. Plug the DIP socket of the glitch stomper into location A9, being careful that the red and blue wires exit to the left rear and that pin number 1 is on the front right.
10. Remove the 74LS02 IC located at A12 and set it aside. Plug the stacked 74LS02-end of the glitch stomper into this socket. Be sure the red and blue wires exit to the left rear and pin 1 is to the front right.
11. Check the pictorial of Fig. 6-6 to be sure you have everything in the right place.
12. Set the cover back in place. Some Apples will have a hook at the right rear that goes into a slot in the case. If yours does, make sure the hook fits into the case slot.
13. Carefully hold case and chassis together and turn them back upside down. **Do not let them separate.**
14. Replace the ten Phillips screws holding case to chassis.
15. Replace all plug-in cards, cables, and whatever.
16. Replace cover, but leave the line cord unplugged till you complete checkout.
17. Label and store your "extra" integrated circuits. Save these should a warranty repair be needed.

INSTRUCTIONS FOR BUILDING THE GLITCH STOMPER

1. Cut the blue wire into two pieces, one $3\frac{3}{4}$ inches long and one $\frac{3}{4}$ inch long. Strip $\frac{3}{16}$ inch of insulation from each end.

Do the same to the red wire.

2. Carefully identify pin 10 of the machined-pin DIP socket. If you have a second 16-pin DIP socket available, plug this machined-pin socket into it. This will keep the pins aligned should the plastic soften. Insert one end of the $3\frac{3}{4}$ -inch blue wire into pin 10 of the DIP socket. Then solder this wire in place.

Position the wire so that it lies flat as shown. Remove the dummy socket if you used one.

3. Carefully identify pin 10 of the 74LS151 1-of-8 data selector. Bend this pin straight out as shown. Tin this pin with a very small amount of solder and, then, solder one end of the $3\frac{3}{4}$ inch red wire to pin 10 of the 74LS151.

Make sure that pin 10 does not point downward or short to adjacent pins.

4. Plug the 74LS151 into the machined-pin DIP socket, making sure that pin 1 of the IC goes to pin 1 of the socket.

Route the wires as shown and temporarily set this half of the modification aside.

5. Press a 74LS02 quad NOR gate into a piece of protective foam. Carefully tin the very tops of pins 7, 10, 12 and 14 with a small amount of solder.

Make sure no solder reaches the part of the pins that must fit the socket.

6. Solder one end of the $\frac{3}{4}$ -inch blue wire to the very top of pin 12 of the 74LS02 as shown.

Then, solder one end of the $\frac{3}{4}$ -inch red wire to the very top of pin 10 of the 74LS02.

Make sure there are no pin-to-pin shorts and that you can still plug this IC into a socket.

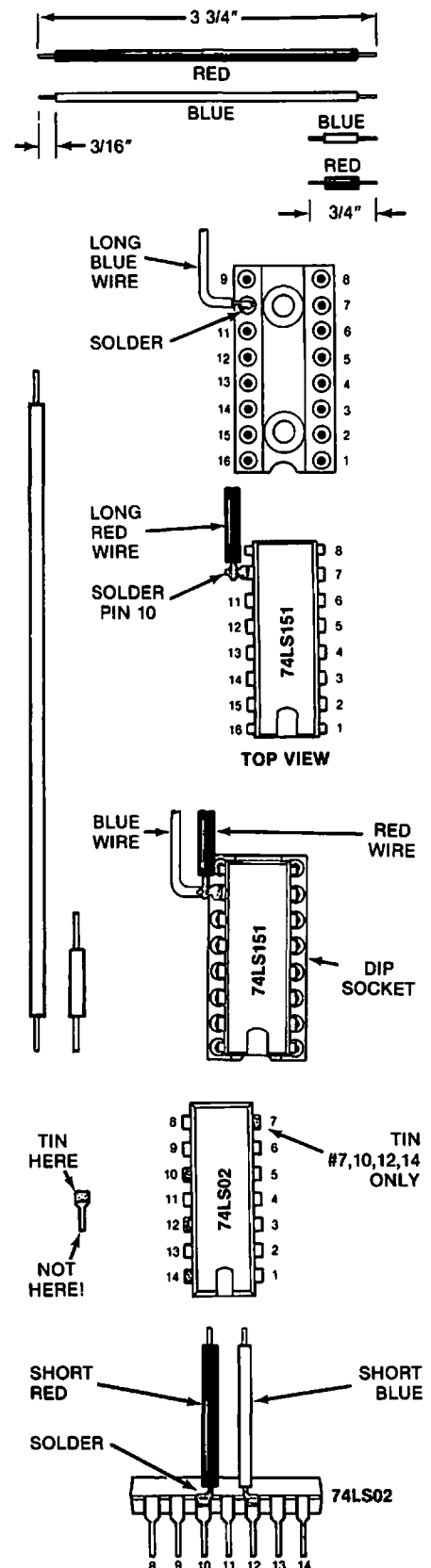
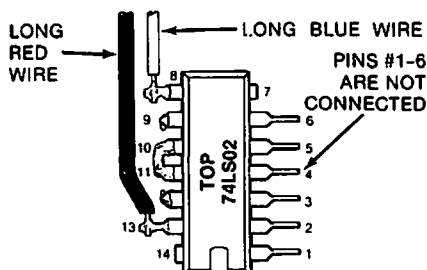
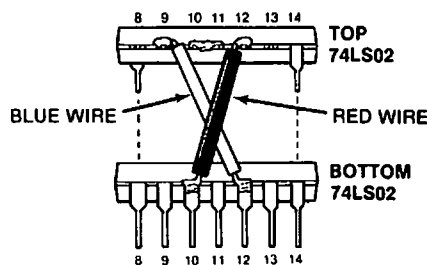
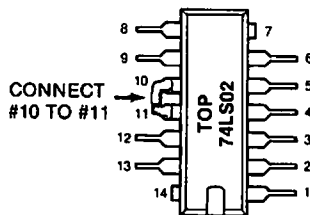
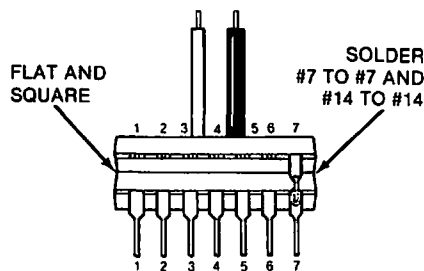
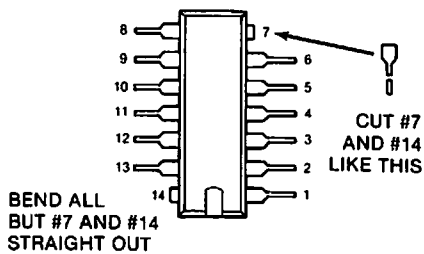


Fig. 6-5. How to build



7. Take a second 74LS02 quad NOR gate and bend pins 1 through 6 straight out. Then, bend pins 8 through 13 straight out.

Then, cut 1/16 inch off the very ends of pins 7 and 14. Tin these pins with a small amount of solder on the cut ends.

8. Piggyback the second 74LS02 quad NOR gate onto the first 74LS02 quad NOR gate so that pins 7 and 14 of each IC contact each other and so that the top IC sits square and flat on the bottom one.

Solder pin 7 to pin 7 and separately solder pin 14 to pin 14. Make sure both ICs "point" the same way and that you can still plug the bottom IC into a socket.

9. On the top 74LS02, bend the tip of pin 11 so that it faces pin 10 and bend the tip of pin 10 so that it faces pin 11.

Solder these two pins together.

10. Take the free end of the short blue wire coming from pin 12 of the bottom 74LS02 and solder this wire to pin 9 of the upper 74LS02 as shown.

Then, take the free end of the short red wire coming from pin 10 of the bottom 74LS02 and solder this to wire to pin 12 of the upper 74LS02 as shown.

Note that the two wires should cross each other, forming an "X".

11. Take the long blue wire coming from the data selector half of your mod and solder this wire to pin 8 of the upper 74LS02, as shown.

Then, take the long red wire coming from the data selector half of your mod and solder this to pin 13 of the upper 74LS02, as shown.

12. Carefully check your glitch stomper against the schematic of Fig. 6-4 and the pictorial of Fig. 6-6.

13. This completes your glitch stomper modification. See text for installation and checkout details.

your glitch stomper.

Here is how to check your glitch stomper modification

GLITCH STOMPER CHECKOUT

1. Turn the Apple OFF and plug in the line cord.
2. Very briefly, turn the Apple on and, then, back off again. The pilot lamp should light and there should be only a single click from the power supply.
3. Turn the Apple on and hit RESET, followed by a few random keys. You should get a text message.
4. Get into BASIC, and, then, type GR. Top of screen should go to black. Then, do a COLOR = 5 followed by a HLIN 0,25 AT 10. You should get a color line on the black screen.
5. Keep typing keys and returns till you get down screen. The usual text should appear in the usual mixed graphics mode.
6. Type HGR. Screen should go black. Type HCOLOR = 3, and, then, HPLOT 0,100 TO 100,0. You should get a single diagonal line on the screen. Check for normal mixed text on the bottom.
7. Run the FUN WITH MIXED FIELDS demo. Everything should work with no glitches.

Enhancement



GENTLE SCROLL

This simple software add-on gives your 48K Apple II a crawling or gentle scroll for easy reading of upward-moving text. The gentle scroll is compatible with the high-resolution character generator.

GENTLE SCROLL

Have you ever been infuriated by how your Apple's text jumps up the screen during a normal scroll? Most personal computers and practically all video terminals share this same hangup. Yet, it is surprisingly simple and easy to add a protected and invisible machine-language subroutine to your 48K Apple II that will give you a smooth and continuous flow of your text up the screen.

I call this enhancement a *gentle scroll*. Fig. 7-1 shows us the differences between a gentle scroll and an abrupt, or ordinary, scroll. In your usual abrupt scroll, the characters move up on the screen an entire character line at a time. Thus, each character dot reappears *eight* dots above where it was before, making it just about impossible to read anything while scrolling. The gentle scroll only moves up *one* dot at a time, giving you the *illusion* of a continuous or crawling text that is very easy to read. During each movement in a gentle scroll, the message only travels one-eighth as far up the screen. Thus, eight movements



In an abrupt or conventional scroll, words move up EIGHT dots at a time. You can not read the screen while it is scrolling.



In a gentle scroll, words move slowly up the screen, ONE dot at a time. You can easily read the screen while it is scrolling. Eight gentle scrolls replace one abrupt scroll.

Fig. 7-1. A gentle scroll is far better looking and easier to read than a conventional or abrupt scroll.

are needed in succession for the same displacement as one old-style abrupt scroll.

You can do a gentle scroll either with hardware or software. I first tried the hardware route and found that elaborate changes in the Apple's system timing would be needed, along with other hassles. Therefore, we will be using a software subroutine instead that works on any 48K, or larger, Apple microcomputer. While this gentle scroll is intended for use in a protected CHARACTER.SET slot under Apple's *HRCC* (High-Res Character Generator), supplied with the *DOS Toolkit*, this gentle scroll will work from most any host program in almost any language. It should be compatible with any other character generator that uses the *HIRES* screen for display.

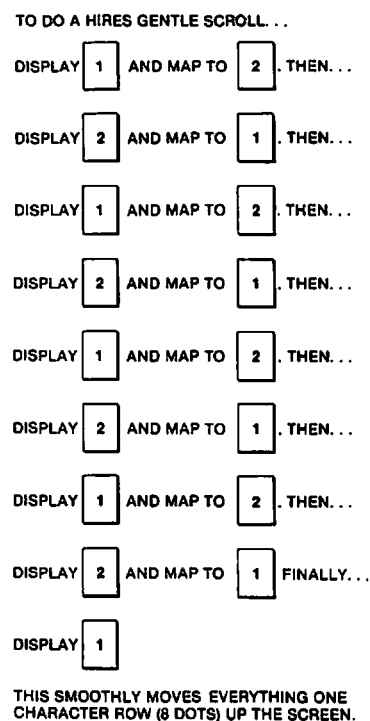
More and more programmers are going the *HIRES* route for character display because of the variety of fonts that you can have, the stunning animation and wipe possibilities, the mixing of graphics and text anywhere on the screen, the wide color range and text-over-color choices, and so on.

Note that our gentle scroll is *only* a gentle scroll. It does not map characters onto the *HIRES* page for you. All the gentle scroll does is move any existing characters or graphics smoothly up the screen. The *HRCC* is an ideal host program to initially enter characters. The *HRCC* interacts with the gentle scroll by replacing its own incredibly ugly abrupt scroll with a subroutine call to the gentle scroll. While you do not have to use the *HRCC*, you will need some other host program to get the characters on the screen and to decide when a scroll is needed.

We will show you a simple Applesoft test program that does not need a character generator. This will get you started but, later on, you will want to add *HRCC* or something similar so as to make best use of your new gentle scroll capability.

Fig. 7-2 shows us how we do a gentle scroll. We use *HIRES* page One for our main text display. Every time we want to do a gentle scroll, the host program (*HRCC*) calls the gentle scroll subroutine. This subroutine then starts mapping from page One to page Two, going up one dot row. Then, we remap from page Two back to page One, going up a second dot row, and so on. We repeat this eight times, ending up back on page One with our text smoothly moved up the screen. Only the page we are mapping *from* ever gets displayed on the screen, so everything appears smooth and continuous. One call to the gentle scrolling

Fig. 7-2. How a gentle scroll is done using pages 1 and 2.



subroutine does all eight mappings needed to smoothly move up one complete row of characters. With the field sync modification of Enhancement 4, the remapping can be made totally invisible. Without this field sync mod, the results are still acceptable but not quite as nice.

While a gentle scroll can be done by remapping a single HIRES page, we have chosen this two-page route for maximum possible speed and smoothness.

THREE PROBLEMS

If a gentle scroll is so fast and easy, why wasn't it available since Year One, and why can't you get it on most other personal computers? It turns out that there are three problems that interact to make a gentle scroll somewhat tricky. These three hassles involve the eye's *perception* of motion, collisions caused by the raster scan, and the *remapping time* needed to get from one HIRES page to another. Let's look at these hassles one by one . . .

Motion perception

This is the simplest of the problems. If you present to the eye two events that are separated by less than 10 milliseconds, both events will appear to exist at the same time. If you take over 100 milliseconds between events, then one event will clearly be seen to happen after the other one, and a distinct jump will be seen between the two.

It is only when you present two events faster than 10 milliseconds, or slower than 100 milliseconds, that the eye "fills in" with the illusion of a smooth and continuous motion. Television uses a 60-field per second rate while most movies use a 48-field per second rate. Both these values center in the range where the eye best senses apparent motion.

The exact speed range over which you can get a smooth illusion varies with the contrast, the image, and many other things, but this 10- to 100-millisecond

area is where we have to aim if we expect to obtain any smooth and useful results.

The Apple presents us with 60 fields per second. This is equal to one field of video every 17 milliseconds. Which tells us that a single scroll should take somewhere between one and six fields to accomplish an action if we are going to get smooth results. We will split the difference and use four fields to remap up one dot line. This should give us the best illusion of an apparently continuous motion.

Our second hassle is called . . .

Raster scan collisions

You've seen this one before. It has ruined more than one animation attempt. There is "sugar," or "collisions," or "flicker," or whatever in the display. Small objects may appear double or may momentarily reverse direction. These distractions can range from just barely noticeable to downright annoying.

On some personal computers, the worst of these distractions are caused by the CPU stealing time from the display for remapping. Fortunately, the Apple has a transparent display that never has to pause to let the microprocessor add bytes to, or remove bytes from, the display memory. Each and every machine cycle on the Apple gets shared 50-50 by the CPU and the display timing. Each works nicely in the other's unneeded and unused blind spot to gain full transparency.

The main cause of sugar or collisions on an Apple display is that the television set paints a raster only one dot at a time. The top part of the raster goes down before the bottom part does. It takes 16 milliseconds to get from the top of the screen to the bottom, giving us some 262 horizontal lines of some 65 microseconds each.

What you can get during animation is a mix of the "old" picture and the "new" picture if you aren't careful. This can range from "just mapped" stuff to old information that is as much as 16 milliseconds out of date. This "old" and "new" mix can momentarily give you a "wrong" display.

Fig. 7-3 shows us how we can get a collision between two character rows if we try moving characters and viewing them at the same time. Most of the time, you either are presenting "old" or "new" information. But, every once in a while, a character will get moved during the time it takes to get from one horizontal line to the next. When this happens on a blank line between characters, the "new" or "dot line" character crashes into the "old" or "upper" dot line without any space between. The result is a brief flash that really can look bad.

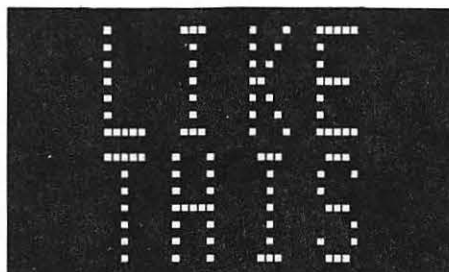
You can also get the opposite effect in which a dot line is dropped out. Instead of a flash, you get an "unflash" in which the middle of an "E," an "H," an "S," or whatever disappears momentarily, leaving you with a strange bunch of illegible dots.

This is sort of like a small town that has one cop and one thief, each of whom are making their rounds. Eventually, they are bound to run into each other even if they are traveling at different speeds and are using different paths going through town.

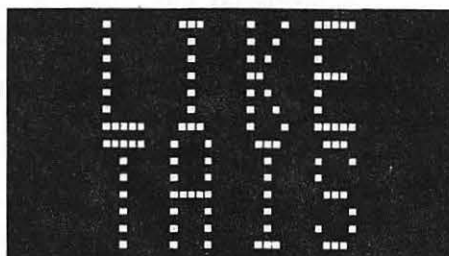
All of which tells us we should never display the HIRES page you are moving characters *to*. While you can do a gentle scroll using a single page of video and remapping things one line at a time, you will find the collisions to be annoying and the motion slightly nervous or erratic. The "one-page" program that I tried turned out to be longer than the available space in one HRCG character set and this is a second disadvantage. You will get the best results by using two HIRES pages for a scroll, displaying one while you move dots to the new location on the other.

For a gentle scroll, both HIRE pages are used alternately to eliminate the "sugar," "sparkle," or "collisions" caused by raster scan problems. These effects usually last one field and can become very annoying.

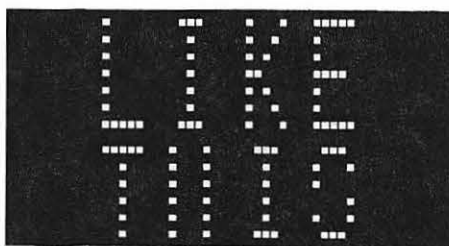
Most of the times, a HIREs map-while-displaying-the-same-page would look...



As the raster scan works its way from top to bottom, it will overrun the mapping every now and then, leading to results...



or perhaps even...



Sugar and collision flashes can be completely eliminated by displaying one HIREs page while you are remapping to the second and, also, by switching between HIREs pages only during the vertical blanking time.

Fig. 7-3. Raster scan collisions.

Even if you switch between the pages after mapping is complete, you can still get a brief flash or collision every now and then. This happens if you decide to switch between pages in the middle of a live scan. Our field sync hardware modification eliminates this final and fairly minor hassle.

We use a program similar to the CRUDE program of Enhancement 4 to make sure we only flip display pages during the vertical blanking time. This way, only "whole" pages appear on the screen and you get no collisions. This particular use of field sync is far less critical than the exact locking needed for mixed fields or other super fancy stuff.

To review, when the field sync modification is in place, we can do a simple bit test of SW3 that waits till it is "safe" to flip from one screen to the other.

The machine-language bit test of location \$C060 sets the negative flag if it is safe to switch the pages and clears the negative flag if it is not. By BPL branching to retest till it is safe, you can always be sure that you will avoid all collisions and all sugar.


What's that location in decimal for use in a BASIC program, you ask? You've got to be kidding. Your Apple warranty isn't long enough to let you do a gentle scroll in BASIC. In fact, unless you are really into machine language, even machine language will seem far too slow. So what you do is provide a special and exceptionally fast machine-language subroutine and call it from any host program in any language you like.

This brings us to the third hassle of . . .

Mapping time

The real sticky problem is that it takes a long time to move all those dots from one HIRES page to a different position on the other one. Let's see. There are 192 rows of 280 dots each for a total of 53,760 dots. We can move these dots seven at a time with a single machine-language byte, so only 7680 moves are needed. But wait. That is 7680 moves to go up a single dot row. We have to go eight dot rows to move up one full character—for a total of 61,440 mappings.

Now, if you use machine language the way most people do, this remapping will take far too long. The usual way we might map is to use the indirect indexed command such as is done in the abrupt scroll included in the HRCG. The code could look something like this . . .

	REMAP	LDA	(P1LOC),Y	; Get a byte from page 1
		STA	(P2LOC),Y	; Move it to page 2
		DEY		; Go to next byte
		BNE	REMAP	; And repeat till done

This says to go to the sum of the address in P1LOC plus the value in the Y register, and put whatever you get from there in the sum of the address in P2LOC plus Y. Both P1LOC and P2LOC will be pairs of page Zero addresses that point to the base address at the left end on any line. Keep doing this remapping till Y hits zero. You start at the right end of one line and work your way to the left, remapping as you go along. The code is short, simple, very powerful, and disgustingly elegant.

It also works, sort of. If you have the time.

The trouble is that this obvious route is far too slow. Let's add things up. We'll assume an Apple machine cycle takes exactly 1 microsecond. If you are a timing purist, just multiply all that follows by 0.978. An indirect load takes 5 microseconds. An indirect store takes 6. Add 2 microseconds to knock one count off Y, and a final 3 microseconds to go back and repeat. A total of 16 microseconds, or just a tad under 16 if you are really keeping score.

Using 16 microseconds to move seven somethings sounds pretty snappy till you realize that we have 61,440 mappings to do. Multiplying 61,440 mappings times 16 microseconds per mapping equals 0.983 second.

We can apparently scroll a row of characters smoothly up eight dots in slightly under a second. This doesn't sound too bad until you realize that there are 24 lines of text on the screen, so it takes nearly 24 seconds to get from the bottom of the screen to the top. This can turn out to be unacceptably slow for some uses.

What can we do to speed up a tight, sophisticated, and elegant little remapping loop like the one we just looked at?


Scrap it.

That's right, scrap our compact and elegant code. Indirect indexed modes dramatically shorten the *physical length* of a program all right, but they take longer to execute. And loops *always* add overhead, since you always have to decrement something and, then, branch to get out of a loop. Even an "empty" loop that does nothing uses up 5 microseconds per trip not doing whatever it is that it is not doing.

The fastest possible code that you could write would use the fastest address mode and no loops. We could write a program that would absolutely load and, then, absolutely store each and every location and it would be much faster. "Brute force" coding of this type would only need 8 microseconds per mapping. Compare this to the 16 microseconds that the elegant code demands.

But, this brute force route would have the disadvantage of needing more than 46,000 bytes of code! And, that's for a single mapping. You have to double that to get back to the first page. Some compromise is obviously needed between code that is short and elegant and code that is long enough to rapidly do the job.

We first notice that absolute indexed addressing is fairly snappy at 4 microseconds per load and 5 microseconds per store. And, we can now use a loop to shorten the code by bunches. But, there's that loop again. The sneaky and crucial trick is to *share* the loop as many ways as possible. If 32 mappings are done inside a loop, the loop overhead of 5 microseconds only takes 5/32 of a microsecond *per mapping*. Our code might look like this . . .

	REMAP	LDA	\$2400,X	; move first byte
		STA	\$4000,X	
		LDA	\$2800,X	; move second byte
		STA	\$4400,X	
		LDA	\$2C00,X	; move third byte
		STA	\$4800,X	
		
		; (repeat 32 times)
		
		LDA	\$3F00,X	; move thirty-second byte
		STA	\$5B00,X	
		DEX		; move one to left
		BNE	REMAP	; and repeat till done

This code is obviously much longer than the earlier code. But look at the timing. Four microseconds to load, five to store, and a mere 5/32 of a microsecond as one mapping's share of the loop overhead. Our total is slightly over nine microseconds, or around one-half the total of using the "elegant and compact" code of the HRCG.

Note that this new code also maps in a different sequence. The usual code works from right to left, one horizontal dot row at a time. Our fast code starts with part of a rightmost column of dots and then works its way from right to left doing a partial column at a time.

The actual mapping details turn out even messier than this, as we will see. The important points here are that an oddball mapping sequence (1) is very fast, and (2) doesn't matter anyway since you don't look at the page till after the mapping is completed. What we are after here is a fast final result, and that is just what we get.

The gentle scroll program turns out fairly long since we are using less elegant addressing modes. The entire program is only 600 bytes long, and fits a protected slot originally intended for use by an alternate character set under HRCG.

When you use the gentle scroll, the measured time on screen is something like 12 seconds from bottom to top. The first time you see this, you will say that it is far too slow. And, if you are listing a program, a gentle scroll may indeed take a minute to list a 100-line program.

Let's look at this 12 seconds in a different light. If we completely fill the screen, we'll average 7 words or so per line. And, 7 words per line times 24 lines equals 168 words in 12 seconds, or a reading speed of 840 words per minute.

This is at least four times faster than most people can read. When you actually use the gentle scroll in your programs, your messages will usually be short. What at first sounds like an awfully slow scroll time actually turns out to be a very useful and attractive speed.

The fast remapping code will take 4 fields to scroll upwards by a single dot, which translates to 32 fields being required in order to do a complete 8-dot full-character scroll. Thus, slightly over half a second is needed for each full-character scrolling.

MEMORY MAP

A memory map of a 48K Apple II, using the gentle scroll under HRCG, is shown in Fig. 7-4. We've already seen back in Enhancement 3 how the bottom 2K-bytes of RAM from \$0000 through \$07FF are reserved for system use. Review the details on this and you will find reference locations on page Zero of memory, the stack on page One, the keyboard buffer on page Two, DOS hooks on page Three, and "page One" of text and low-resolution graphics on memory pages Four through Seven. Here a memory "page" consists of 256 locations, while a video "page" is made up of whatever happens to fit the screen. One 1024-byte text or LORES page needs four RAM memory pages of 256 bytes each.

Looking further up the map, we see that HIRES page One fits in locations \$2000 through \$3FFF, and HIRES page Two resides at hex \$4000 through \$5FFF. Each HIRES page needs 8K of RAM to store its image. Thus, thirty-two pages of RAM memory will always be needed for each HIRES page.

DOS normally resides at the top of RAM, and goes from locations \$9600 up through \$BFFF. The HRCG high-resolution character generator and its alternate character sets reside just under DOS.

The HRCG is an example of a new type of program that uses an "R" disk file. The "R" stands for *relocatable*, and this coding must be handled differently from the usual "A," "B," "I," or "T" files you already know. When you use the LOADHRCG program off the *DOS Toolkit*, the HRCG is automatically put just below DOS and, then, enough space is cleared below HRCG for as many alternate fonts as are needed. After enough room is set aside and the fonts are loaded, the Applesoft HIMEM pointer is then automatically moved down. This will protect the HRCG and any of the alternate character sets from intrusion.

This nicely eliminates the problem of making room for machine-language sequences with Applesoft. Just put your sequence in the space intended for the highest HRCG alternate character set, and the routine gets put out of harm's way. Everything is set aside for you automatically without any calculations or sneaky tricks.

You'll find two areas left in the memory map for your Applesoft or other program. The 6K space from \$800 through \$2000 is available, as are the 10K+ locations \$6000 through \$8AFE.

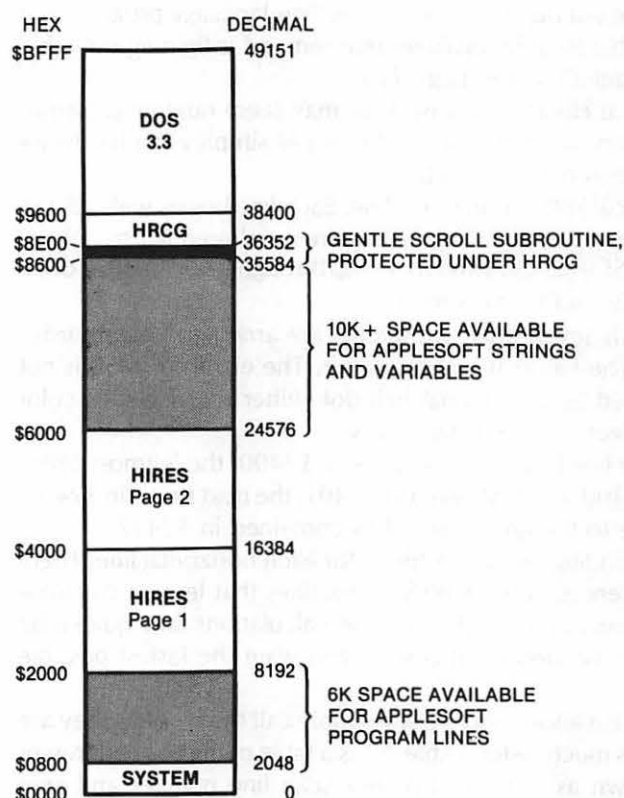


Fig. 7-4. Memory map of 48K RAM under gentle scroll. Most shorter Applesoft programs will fit as is. Longer ones may need pointer adjustment, disk reaccess, RAM card use or other memory management.

Applesoft normally builds its program up from its TXTAB pointer at the lowest available RAM location. This is usually \$0800. It also builds its variables down from HIMEM. You will usually have around 6K of program space and some 10K of variable space available till you bump into either HIRES page.

Most shorter Applesoft programs will fit as is. Longer ones may need pointer adjustment, disk reaccess, REM elimination, or other memory management tricks. Note that this is nearly the same problem that you have any time you want to use both HIRES pages and Applesoft. The HRCG and its alternate fonts take up little extra room.

Our gentle scroll program should get loaded as the *highest* character set under the HRCG. But, we will never use this set as a character set. Instead, we will find the magic hooks to jump to this code subroutine every time the HRCG wants to do a scroll. The reason we use a character set location is for the convenience it gives us in automatically loading a machine-language program into a protected space. On a 48K Apple computer operating under DOS 3.3, the highest character set lies from \$8AFF through \$8CFE. If you do use any alternate character sets, they will automatically be placed lower in memory. Regardless of how many alternate fonts you use, the highest character set will always start at \$8AFF. Should you use alternate character sets, each additional character set will need another 3 pages, working down through memory.

Thus, the highest alternate font will always lie from \$8AFF to \$8DFF, and the lowest alternate font will always have the lowest starting address, regardless of how many fonts you use. All of these will reside in a protected space above HIMEM.

We now know where to put our gentle scroll machine-language program and how to get it there. One big location problem that remains is figuring out what maps where on HIRES page One and page Two.

The first time you look at HIRES locations, they may seem random or senseless. In fact, they were very carefully chosen to greatly simplify the hardware needed in the Apple's video timing circuitry.

Fig. 7-5 shows us a typical HIRES horizontal line. Each line begins with a BASE address on the left and needs 40 bytes. The bytes are numbered left to right as BASE + 0, BASE + 1, BASE + 2, . . . , and so on, up through BASE + 39 in decimal, or BASE + \$27 in hex, at the extreme right.

Each byte, in turn, holds seven dots. These dots are arranged "backwards" as the lower seven bits. The LSB is the *leftmost* dot. The eighth or MSB is not mapped. Instead, it is used as an optional half-dot shifter that gives us color changes by shifting all seven dots simultaneously.

As an example, if some line has a base address of \$2400, the leftmost seven dots will be in location \$2400, the next seven in \$2401, the next seven in \$2402, and so on across the line to the final seven dots contained in \$2427.

What gets sticky fast is finding the base address for each horizontal line. There are elegant program sequences in the HIRES subroutines that let you calculate each and every base address as needed. But these calculations take quite a bit of time and they must be avoided at all costs if you want the fastest possible code.

Rather than calculate base addresses, we will simply call them out as they are needed in the code. This is much faster. Table 7-1 is a table of the base addresses of each line. This is shown as a decimal or hex scan line number and as a "which-dot-row-of-which-character" number pair.

The base addresses are rearranged for you in Fig. 7-6. Here we see the packing of lines into each sequential 256-byte page of memory. These addresses are all shown for HIRES page One on the left and HIRES page Two on the right. To find a comparable address for Hires page Two, just add hex \$2000 to each page One location. To go the other way, subtract hex \$2000.

Now, if we were trying to display and remap on the same page at the same time, we would have to start at the top of the display and remap each line one place up from where it happened to be. This would take 192 separate remappings since there are 192 vertical lines (24 characters \times 8 dots per character)

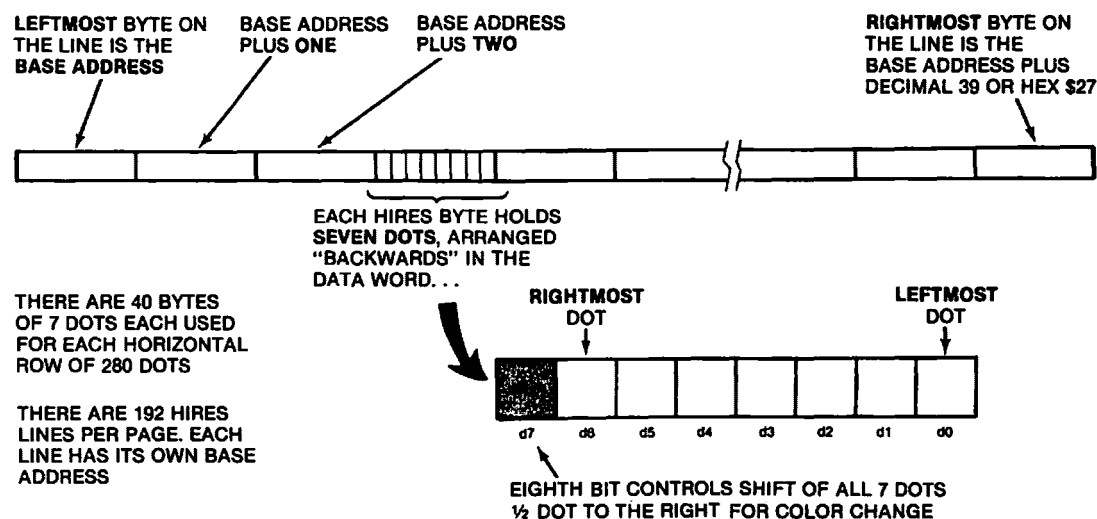


Fig. 7-5. Details of one horizontal line mapping.

	\$00	\$28	\$50	\$80	\$A8	\$D0	\$FF	
\$2000	0	64	128	8	72	136		\$4000
\$2100	16	80	144	24	88	152		\$4100
\$2200	32	96	160	40	104	168		\$4200
\$2300	48	112	176	56	120	184		\$4300
\$2400	1	65	129	9	73	137		\$4400
\$2500	17	81	145	25	89	153		\$4500
\$2600	33	97	161	41	105	169		\$4600
\$2700	49	113	177	57	121	185		\$4700

	\$00	\$28	\$50	\$80	\$A8	\$D0	\$FF	
\$2800	2	66	130	10	74	138		\$4800
\$2900	18	82	146	26	90	154		\$4900
\$2A00	34	98	162	42	106	170		\$4A00
\$2B00	50	114	178	58	122	186		\$4B00
\$2C00	3	67	131	11	75	139		\$4C00
\$2D00	19	83	147	27	91	155		\$4D00
\$2E00	35	99	163	43	107	171		\$4E00
\$2F00	51	115	179	59	123	187		\$4F00

	\$00	\$28	\$50	\$80	\$A8	\$D0	\$FF	
\$3000	4	68	132	12	76	140		\$5000
\$3100	20	84	148	28	92	156		\$5100
\$3200	36	100	164	44	108	172		\$5200
\$3300	52	116	180	60	124	188		\$5300
\$3400	5	69	133	13	77	141		\$5400
\$3500	21	85	149	29	93	157		\$5500
\$3600	37	101	165	45	109	173		\$5600
\$3700	53	117	181	61	125	189		\$5700

	\$00	\$28	\$50	\$80	\$A8	\$D0	\$FF	
\$3800	6	70	134	14	78	142		\$5800
\$3900	22	86	150	30	94	158		\$5900
\$3A00	38	102	166	46	110	174		\$5A00
\$3B00	54	118	182	62	126	190		\$5B00
\$3C00	7	71	135	15	79	143		\$5C00
\$3D00	23	87	151	31	95	159		\$5D00
\$3E00	39	103	167	47	111	175		\$5E00
\$3F00	55	119	183	63	127	191		\$5F00

	\$00	\$28	\$50	\$80	\$A8	\$D0	\$FF	
--	------	------	------	------	------	------	------	--

EACH BLOCK SHOWS HORIZONTAL LINE NUMBER
IN DECIMAL. ■ = UNUSED 8 BYTES OF RAM.

Fig. 7-6. How HIRES lines are packed into memory.

in the display. We would map line 1 to line 0, then line 2 to line 1, and so on down the screen.

But, a two-page mapping lets us get sneaky and greatly shorten the code. An indexed move lets us move up to 256 bytes from one base address. Do things just right and we can be remapping *six* lines using the same indexed move instruction. This shortens the code bunched. For instance, one indexed mapping code pair can map lines 1 to 0, 65 to 64, 129 to 128, 9 to 8, 73 to 72, and 137 to 136. All of this is done with the same base address of hex \$2400. You first set your X index pointer to \$F7 and work your way down the list. First, you remap all of line 137, then all of line 73, then all of line 9. Then, you reset your

Table 7-1. HIRES BASE ADDRESSES

LINE NUMBER			PAGE 1 BASE		PAGE 2 BASE	
DECIMAL	CHAR/DOT	HEX	DECIMAL	HEX	DECIMAL	HEX
0	0/0	\$00	8192	\$2000	16384	\$4000
1	0/1	\$01	9216	\$2400	17408	\$4400
2	0/2	\$02	10240	\$2800	18432	\$4800
3	0/3	\$03	11264	\$2C00	19456	\$4C00
4	0/4	\$04	12288	\$3000	20480	\$5000
5	0/5	\$05	13312	\$3400	21504	\$5400
6	0/6	\$06	14336	\$3800	22528	\$5800
7	0/7	\$07	15360	\$3C00	23552	\$5C00
8	1/0	\$08	8320	\$2080	16512	\$4080
9	1/1	\$09	9344	\$2480	17536	\$4480
10	1/2	\$0A	10368	\$2880	18560	\$4880
11	1/3	\$0B	11392	\$2C80	19584	\$4C80
12	1/4	\$0C	12416	\$3080	20608	\$5080
13	1/5	\$0D	13440	\$3480	21632	\$5480
14	1/6	\$0E	14464	\$3880	22656	\$5880
15	1/7	\$0F	15488	\$3C80	23680	\$5C80
16	2/0	\$10	8448	\$2100	16640	\$4100
17	2/1	\$11	9472	\$2500	17664	\$4500
18	2/2	\$12	10496	\$2900	18688	\$4900
19	2/3	\$13	11520	\$2D00	19712	\$4D00
20	2/4	\$14	12544	\$3100	20736	\$5100
21	2/5	\$15	13568	\$3500	21760	\$5500
22	2/6	\$16	14592	\$3900	22784	\$5900
23	2/7	\$17	15616	\$3D00	23808	\$5D00
24	3/0	\$18	8576	\$2180	16768	\$4180
25	3/1	\$19	9600	\$2580	17792	\$4580
26	3/2	\$1A	10624	\$2980	18816	\$4980
27	3/3	\$1B	11648	\$2D80	19840	\$4D80
28	3/4	\$1C	12672	\$3180	20864	\$5180
29	3/5	\$1D	13696	\$3580	21888	\$5580
30	3/6	\$1E	14720	\$3980	22912	\$5980
31	3/7	\$1F	15744	\$3D80	23936	\$5D80
32	4/0	\$20	8704	\$2200	16896	\$4200
33	4/1	\$21	9728	\$2600	17920	\$4600
34	4/2	\$22	10752	\$2A00	18944	\$4A00
35	4/3	\$23	11776	\$2E00	19968	\$4E00
36	4/4	\$24	12800	\$3200	20992	\$5200
37	4/5	\$25	13824	\$3600	22016	\$5600
38	4/6	\$26	14848	\$3A00	23040	\$5A00
39	4/7	\$27	15872	\$3E00	24064	\$5E00
40	5/0	\$28	8832	\$2280	17024	\$4280
41	5/1	\$29	9856	\$2680	18048	\$4680
42	5/2	\$2A	10880	\$2A80	19072	\$4A80
43	5/3	\$2B	11904	\$2E80	20096	\$4E80
44	5/4	\$2C	12928	\$3280	21120	\$5280
45	5/5	\$2D	13952	\$3680	22144	\$5680
46	5/6	\$2E	14976	\$3A80	23168	\$5A80
47	5/7	\$2F	16000	\$3E80	24192	\$5E80
48	6/0	\$30	8960	\$2300	17152	\$4300
49	6/1	\$31	9984	\$2700	18176	\$4700
50	6/2	\$32	11008	\$2B00	19200	\$4B00
51	6/3	\$33	12032	\$2F00	20224	\$4F00
52	6/4	\$34	13056	\$3300	21248	\$5300
53	6/5	\$35	14080	\$3700	22272	\$5700
54	6/6	\$36	15104	\$3B00	23296	\$5B00
55	6/7	\$37	16128	\$3F00	24320	\$5F00

Table 7-1 Cont. HIRES BASE ADDRESSES

LINENUMBER		HEX	PAGE 1 BASE		PAGE 2 BASE	
DECIMAL	CHAR/DOT		DECIMAL	HEX	DECIMAL	HEX
56	7/0	\$38	9088	\$2380	17280	\$4380
57	7/1	\$39	10112	\$2780	18304	\$4780
58	7/2	\$3A	11136	\$2B80	19328	\$4B80
59	7/3	\$3B	12160	\$2F80	20352	\$4F80
60	7/4	\$3C	13184	\$3380	21376	\$5380
61	7/5	\$3D	14208	\$3780	22400	\$5780
62	7/6	\$3E	15232	\$3B80	23424	\$5B80
63	7/7	\$3F	16256	\$3F80	24448	\$5F80
64	8/0	\$40	8232	\$2028	16424	\$4028
65	8/1	\$41	9256	\$2428	17448	\$4428
66	8/2	\$42	10280	\$2828	18472	\$4828
67	8/3	\$43	11304	\$2C28	19496	\$4C28
68	8/4	\$44	12328	\$3028	20520	\$5028
69	8/5	\$45	13352	\$3428	21544	\$5428
70	8/6	\$46	14376	\$3828	22568	\$5828
71	8/7	\$47	15400	\$3C28	23592	\$5C28
72	9/0	\$48	8360	\$20A8	16552	\$40A8
73	9/1	\$49	9384	\$24A8	17576	\$44A8
74	9/2	\$4A	10408	\$28A8	18600	\$48A8
75	9/3	\$4B	11432	\$2CA8	19624	\$4CA8
76	9/4	\$4C	12456	\$30A8	20648	\$50A8
77	9/5	\$4D	13480	\$34A8	21672	\$54A8
78	9/6	\$4E	14504	\$38A8	22696	\$58A8
79	9/7	\$4F	15528	\$3CA8	23720	\$5CA8
80	10/0	\$50	8488	\$2128	16680	\$4128
81	10/1	\$51	9512	\$2528	17704	\$4528
82	10/2	\$52	10536	\$2928	18728	\$4928
83	10/3	\$53	11560	\$2D28	19752	\$4D28
84	10/4	\$54	12584	\$3128	20776	\$5128
85	10/5	\$55	13608	\$3528	21800	\$5528
86	10/6	\$56	14632	\$3928	22824	\$5928
87	10/7	\$57	15656	\$3D28	23848	\$5D28
88	11/0	\$58	8616	\$21A8	16808	\$41A8
89	11/1	\$59	9640	\$25A8	17832	\$45A8
90	11/2	\$5A	10664	\$29A8	18856	\$49A8
91	11/3	\$5B	11688	\$2DA8	19880	\$4DA8
92	11/4	\$5C	12712	\$31A8	20904	\$51A8
93	11/5	\$5D	13736	\$35A8	21928	\$55A8
94	11/6	\$5E	14760	\$39A8	22952	\$59A8
95	11/7	\$5F	15784	\$3DA8	23976	\$5DA8
96	12/0	\$60	8744	\$2228	16936	\$4228
97	12/1	\$61	9768	\$2628	17960	\$4628
98	12/2	\$62	10792	\$2A28	18984	\$4A28
99	12/3	\$63	11816	\$2E28	20008	\$4E28
100	12/4	\$64	12840	\$3228	21032	\$5228
101	12/5	\$65	13864	\$3628	22056	\$5628
102	12/6	\$66	14888	\$3A28	23080	\$5A28
103	12/7	\$67	15912	\$3E28	24104	\$5E28
104	13/0	\$68	8872	\$22A8	17064	\$42A8
105	13/1	\$69	9896	\$26A8	18088	\$46A8
106	13/2	\$6A	10920	\$2AA8	19112	\$4AA8
107	13/3	\$6B	11944	\$2EA8	20136	\$4EA8
108	13/4	\$6C	12968	\$32A8	21160	\$52A8
109	13/5	\$6D	13992	\$36A8	22184	\$56A8
110	13/6	\$6E	15016	\$3AA8	23208	\$5AA8
111	13/7	\$6F	16040	\$3EA8	24232	\$5EA8

Table 7-1 Cont. HIRES BASE ADDRESSES

LINE NUMBER		HEX	PAGE 1 BASE		PAGE 2 BASE	
DECIMAL	CHAR/DOT		DECIMAL	HEX	DECIMAL	HEX
112	14/0	\$70	9000	\$2328	17192	\$4328
113	14/1	\$71	10024	\$2728	18216	\$4728
114	14/2	\$72	11048	\$2B28	19240	\$4B28
115	14/3	\$73	12072	\$2F28	20264	\$4F28
116	14/4	\$74	13096	\$3328	21288	\$5328
117	14/5	\$75	14120	\$3728	22312	\$5728
118	14/6	\$76	15144	\$3B28	23336	\$5B28
119	14/7	\$77	16168	\$3F28	24360	\$5F28
120	15/0	\$78	9128	\$23A8	17320	\$43A8
121	15/1	\$79	10152	\$27A8	18344	\$47A8
122	15/2	\$7A	11176	\$2BA8	19368	\$4BA8
123	15/3	\$7B	12200	\$2FA8	20392	\$4FA8
124	15/4	\$7C	13224	\$33A8	21416	\$53A8
125	15/5	\$7D	14248	\$37A8	22440	\$57A8
126	15/6	\$7E	15272	\$3BA8	23464	\$5BA8
127	15/7	\$7F	16296	\$3FA8	24488	\$5FA8
128	16/0	\$80	8272	\$2050	16464	\$4050
129	16/1	\$81	9296	\$2450	17488	\$4450
130	16/2	\$82	10320	\$2850	18512	\$4850
131	16/3	\$83	11344	\$2C50	19536	\$4C50
132	16/4	\$84	12368	\$3050	20560	\$5050
133	16/5	\$85	13392	\$3450	21584	\$5450
134	16/6	\$86	14416	\$3850	22608	\$5850
135	16/7	\$87	15440	\$3C50	23632	\$5C50
136	17/0	\$88	8400	\$20D0	16592	\$40D0
137	17/1	\$89	9424	\$24D0	17616	\$44D0
138	17/2	\$8A	10448	\$28D0	18640	\$48D0
139	17/3	\$8B	11472	\$2CD0	19664	\$4CD0
140	17/4	\$8C	12496	\$30D0	20688	\$50D0
141	17/5	\$8D	13520	\$34D0	21712	\$54D0
142	17/6	\$8E	14544	\$38D0	22736	\$58D0
143	17/7	\$8F	15568	\$3CD0	23760	\$5CD0
144	18/0	\$90	8528	\$2150	16720	\$4150
145	18/1	\$91	9552	\$2550	17744	\$4550
146	18/2	\$92	10576	\$2950	18768	\$4950
147	18/3	\$93	11600	\$2D50	19792	\$4D50
148	18/4	\$94	12624	\$3150	20816	\$5150
149	18/5	\$95	13648	\$3550	21840	\$5550
150	18/6	\$96	14672	\$3950	22864	\$5950
151	18/7	\$97	15696	\$3D50	23888	\$5D50
152	19/0	\$98	8656	\$21D0	16848	\$41D0
153	19/1	\$99	9680	\$25D0	17872	\$45D0
154	19/2	\$9A	10704	\$29D0	18896	\$49D0
155	19/3	\$9B	11728	\$2DD0	19920	\$4DD0
156	19/4	\$9C	12752	\$31D0	20944	\$51D0
157	19/5	\$9D	13776	\$35D0	21968	\$55D0
158	19/6	\$9E	14800	\$39D0	22992	\$59D0
159	19/7	\$9F	15824	\$3DD0	24016	\$5DD0
160	20/0	\$A0	8784	\$2250	16976	\$4250
161	20/1	\$A1	9808	\$2650	18000	\$4650
162	20/2	\$A2	10832	\$2A50	19024	\$4A50
163	20/3	\$A3	11856	\$2E50	20048	\$4E50
164	20/4	\$A4	12880	\$3250	21072	\$5250
165	20/5	\$A5	13904	\$3650	22096	\$5650
166	20/6	\$A6	14928	\$3A50	23120	\$5A50
167	20/7	\$A7	15952	\$3E50	24144	\$5E50

Table 7-1 Cont. HIRES BASE ADDRESSES

LINE NUMBER			PAGE 1 BASE		PAGE 2 BASE	
DECIMAL	CHAR/DOT	HEX	DECIMAL	HEX	DECIMAL	HEX
168	21/0	\$A8	8912	\$22D0	17104	\$42D0
169	21/1	\$A9	9936	\$26D0	18128	\$46D0
170	21/2	\$AA	10960	\$2AD0	19152	\$4AD0
171	21/3	\$AB	11984	\$2ED0	20176	\$4ED0
172	21/4	\$AC	13008	\$32D0	21200	\$52D0
173	21/5	\$AD	14032	\$36D0	22224	\$56D0
174	21/6	\$AE	15056	\$3AD0	23248	\$5AD0
175	21/7	\$AF	16080	\$3ED0	24272	\$5ED0
176	22/0	\$B0	9040	\$2350	17232	\$4350
177	22/1	\$B1	10064	\$2750	18256	\$4750
178	22/2	\$B2	11088	\$2B50	19280	\$4B50
179	22/3	\$B3	12112	\$2F50	20304	\$4F50
180	22/4	\$B4	13136	\$3350	21328	\$5350
181	22/5	\$B5	14160	\$3750	22352	\$5750
182	22/6	\$B6	15184	\$3B50	23376	\$5B50
183	22/7	\$B7	16208	\$3F50	24400	\$5F50
184	23/0	\$B8	9168	\$23D0	17360	\$43D0
185	23/1	\$B9	10192	\$27D0	18384	\$47D0
186	23/2	\$BA	11216	\$2BD0	19408	\$4BD0
187	23/3	\$BB	12240	\$2FD0	20432	\$4FD0
188	23/4	\$BC	13264	\$33D0	21456	\$53D0
189	23/5	\$BD	14288	\$37D0	22480	\$57D0
190	23/6	\$BE	15312	\$3BD0	23504	\$5BD0
191	23/7	\$BF	16336	\$3FD0	24528	\$5FD0

X index to hex \$77 and work down all of line 129, followed by all of line 65 and, finally, all of line 1. Trace this action out on Fig. 7-6 to make sure you see and understand exactly what is happening.

However, remapping several lines with the same code does give you several minor hassles that you have to get around. Those unused locations from \$78 through \$7F and \$F8 through \$FF on any page must be bypassed during the mapping. Besides making the code take longer, a glitch or two will remain on the screen if you do not carefully bypass all unused locations.

As an aside, note that these unused locations are usually plowed anytime you clear the HIRES screen. Thus, while these "free" locations are just sitting there, you can't safely use them for anything. This is in contrast to the 64 bytes hidden on the page 1 text and LORES1 screen which are used for I/O and are very carefully protected during a screen clear.

There are also a few lines that do not remap exactly in the way you might expect them to. These lines must each be "custom" mapped because they cause a move from column to column in Fig. 7-6. Five of the special cases are 64 to 63, 128 to 127, 8 to 7, 72 to 71, and 136 to 135. There is a sixth special case of 0 to nowhere, but since nowhere is off screen, we can ignore this mapping. Again, check into Fig. 7-6 to find out why these five mappings are special.

PROGRAM AND FLOWCHART

A flowchart of the gentle scroll program is shown in Fig. 7-7, while Program 7-1 gives you an assembler listing of the GENTLE SCROLL.SET program. Fig. 7-8 shows a hex dump of the GENTLE SCROLL.SET. Details for copying and using the GENTLE SCROLL.SET program follow:

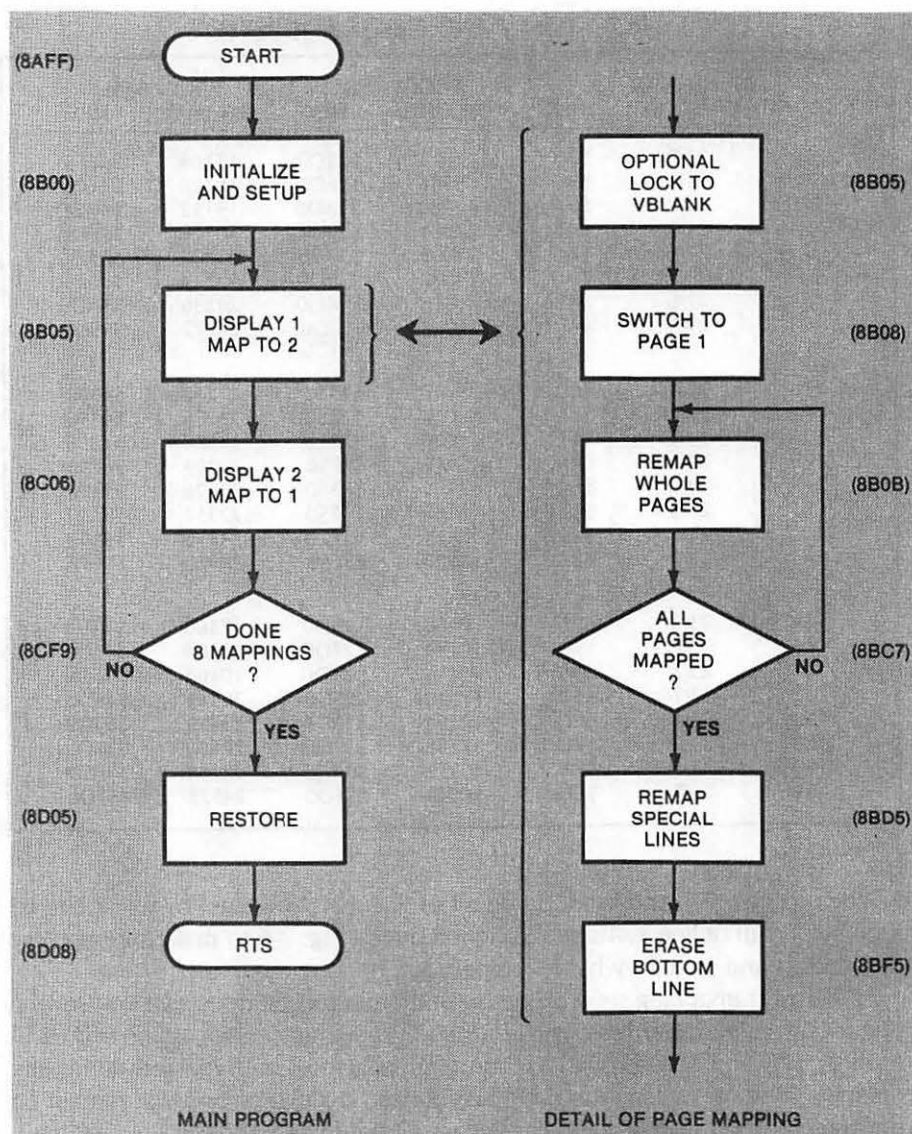


Fig. 7-7. Flowchart of GENTLE SCROLLSET. Machine-language subroutine is called whenever an 8-dot gentle scroll of HIRES page 1 is wanted.

We begin by saving all registers and, then, going to SYNC subroutine that optionally waits for the vertical blanking time before continuing. Page One is then displayed. We then remap most of the lines on page One onto page Two, moving each line up one dot row. We start at the right of the screen and make our way rapidly down and slowly to the left. This oddball scheme lets us share our loop timing 31 ways and it doesn't matter anyway since we aren't looking at what we are mapping till we are finished.

Unused locations \$F8 through \$FF are never mapped. We ignore these by beginning with an index value of \$F7 and working down. Unused locations \$78 through \$7F are bypassed with a compare and fix. Should we get to an index value of \$7F, this value is immediately changed to \$77, thus bypassing these unused locations.

The bulk of the mapping is done six lines at a time, handling a total of $31 \times 6 = 186$ lines.

After the bulk of the mapping is complete, we then custom handle the five special cases. These get remapped individually.

PROGRAM 7-1

GENTLE SCROLL SET

LANGUAGE: APPLE ASSEMBLER

NEEDS: HIRES SUPPORT PROGRAM
SUCH AS HRCG.
HIMEM < 35583

```

8AFF:      5 ; *****
8AFF:      6 ; *
8AFF:      7 ; *      GENTLE      *
8AFF:      8 ; *      SCROLL.SET  *
8AFF:      9 ; *
8AFF:     10 ; *      VERSION  3.1  *
8AFF:     11 ; *      ( 3-18-81)  *
8AFF:     12 ; *
8AFF:     13 ; *      COPYRIGHT 1981 *
8AFF:     14 ; *
8AFF:     15 ; * BY DON LANCASTER *
8AFF:     16 ; * AND SYNERGETICS  *
8AFF:     17 ; *
8AFF:     18 ; *      ALL COMMERCIAL *
8AFF:     19 ; *      RIGHTS RESERVED *
8AFF:     20 ; *
8AFF:     21 ; *****

8AFF:     23 ; THIS PROGRAM CREATES A PAGE
8AFF:     24 ; ONE GENTLE SCROLL UNDER HRCG
8AFF:     25 ; FOR USE ON 48K APPLE II.

8AFF:     27 ; IT LOADS AS THE HIGHEST CHARACTER
8AFF:     28 ; SET AND USES HIRES PAGE TWO
8AFF:     29 ; FOR A WORKSPACE.

8AFF:     31 ; A VSYNC HARDWARE MOD CONNECTING
8AFF:     32 ; 4/C14 TO 4/H14 IS RECOMMENDED
8AFF:     33 ; FOR BEST OPERATION.

8AFF:     35 ; IF YOU ARE NOT USING HRCG --

8AFF:     37 ; A MACHINE JSR $8AFF -OR-
8AFF:     38 ; AN INTEGER CALL -29953 -OR-
8AFF:     39 ; AN APPLESOFT CALL 35583
8AFF:     40 ; GIVES YOU AN EIGHT SCAN LINE
8AFF:     41 ; GENTLE SCROLL ON HIRES 1

8AFF:     43 ; PROTECT THIS SPACE WITH
8AFF:     44 ; AN APPLESOFT HIMEM:35582.

```

PROGRAM 7-1, CONT'D...

```
8AFF:      46 ;   IF YOU ARE USING HRCG AND
8AFF:      47 ;   MACHINE LANGUAGE --

8AFF:      49 ;       GENTLE SCROLL ON -- $9214: 00 8B
8AFF:      50 ;       GENTLE SCROLL OFF - $9214: 21 92

8AFF:      52 ;   IF YOU ARE USING HRCG AND
8AFF:      53 ;   APPLESOFT --

8AFF:      55 ;       GENTLE SCROLL ON -- POKE 37396,0
8AFF:      56 ;               AND -- POKE 37397,139

8AFF:      58 ;       GENTLE SCROLL OFF - POKE 37396,33
8AFF:      59 ;               AND -- POKE 37397,146
8AFF:      60 ;       POKE ONLY AT TOP
8AFF:      61 ;       OF SCREEN!
```

PROGRAM 7-1, CONT'D...

8AFF: 65 ; **** HOOKS ****

FF3F: 67 IOREST EQU \$FF3F
 FF4A: 68 IOSAVE EQU \$FF4A
 C054: 69 PAGE1 EQU \$C054
 C055: 70 PAGE2 EQU \$C055
 C060: 71 VSYNC EQU \$C060 ; 4/C14 VIA 4/H14 SYNC MOD

8AFF: 73 ; ***** MAIN PROGRAM *****

8AFF:EA 75 NOP ; ADJUST RELOCATE START
 8B00:20 4A FF 76 START JSR IOSAVE ; SAVE ALL REGISTERS

 8B03:A0 04 78 LDY #\$04 ; FOR FOUR PAIRS OF MAPPINGS
 8B05:20 09 8D 79 GOODD JSR SYNC ; OPTIONAL LOCK TO VBLANK
 8B08:8E 54 C0 80 STX PAGE1 ; DISPLAY PAGE ONE

 8B0B:A2 F7 82 LDX #\$F7 ; BYPASS UNMAPPED BITS
 8B0D:BD 00 24 83 NXODD LDA \$2400,X ; LINES 1-65-129-9-...
 8B10:9D 00 40 84 STA \$4000,X
 8B13:BD 00 28 85 LDA \$2800,X
 8B16:9D 00 44 86 STA \$4400,X
 8B19:BD 00 2C 87 LDA \$2C00,X
 8B1C:9D 00 48 88 STA \$4800,X
 8B1F:BD 00 30 89 LDA \$3000,X
 8B22:9D 00 4C 90 STA \$4C00,X
 8B25:BD 00 34 91 LDA \$3400,X
 8B28:9D 00 50 92 STA \$5000,X
 8B2B:BD 00 38 93 LDA \$3800,X
 8B2E:9D 00 54 94 STA \$5400,X
 8B31:BD 00 3C 95 LDA \$3C00,X
 8B34:9D 00 58 96 STA \$5800,X

 8B37:BD 00 21 98 LDA \$2100,X ; LINES 16-80-144-24-...
 8B3A:9D 80 5C 99 STA \$5C80,X
 8B3D:BD 00 25 100 LDA \$2500,X
 8B40:9D 00 41 101 STA \$4100,X
 8B43:BD 00 29 102 LDA \$2900,X
 8B46:9D 00 45 103 STA \$4500,X
 8B49:BD 00 2D 104 LDA \$2D00,X
 8B4C:9D 00 49 105 STA \$4900,X
 8B4F:BD 00 31 106 LDA \$3100,X
 8B52:9D 00 4D 107 STA \$4D00,X
 8B55:BD 00 35 108 LDA \$3500,X
 8B58:9D 00 51 109 STA \$5100,X
 8B5B:BD 00 39 110 LDA \$3900,X
 8B5E:9D 00 55 111 STA \$5500,X
 8B61:BD 00 3D 112 LDA \$3D00,X
 8B64:9D 00 59 113 STA \$5900,X

PROGRAM 7-1, CONT'D...

```

8B67:BD 00 22 115      LDA  $2200,X      ; LINES 32-96-160-40-...
8B6A:9D 80 5D 116      STA  $5D80,X
8B6D:BD 00 26 117      LDA  $2600,X
8B70:9D 00 42 118      STA  $4200,X
8B73:BD 00 2A 119      LDA  $2A00,X
8B76:9D 00 46 120      STA  $4600,X
8B79:BD 00 2E 121      LDA  $2E00,X
8B7C:9D 00 4A 122      STA  $4A00,X
8B7F:BD 00 32 123      LDA  $3200,X
8B82:9D 00 4E 124      STA  $4E00,X
8B85:BD 00 36 125      LDA  $3600,X
8B88:9D 00 52 126      STA  $5200,X
8B8B:BD 00 3A 127      LDA  $3A00,X
8B8E:9D 00 56 128      STA  $5600,X
8B91:BD 00 3E 129      LDA  $3E00,X
8B94:9D 00 5A 130      STA  $5A00,X

8B97:BD 00 23 132      LDA  $2300,X      ; LINES 48-112-176-56-...
8B9A:9D 80 5E 133      STA  $5E80,X
8B9D:BD 00 27 134      LDA  $2700,X
8BA0:9D 00 43 135      STA  $4300,X
8BA3:BD 00 2B 136      LDA  $2B00,X
8BA6:9D 00 47 137      STA  $4700,X
8BA9:BD 00 2F 138      LDA  $2F00,X
8BAC:9D 00 4B 139      STA  $4B00,X
8BAF:BD 00 33 140      LDA  $3300,X
8BB2:9D 00 4F 141      STA  $4F00,X
8BB5:BD 00 37 142      LDA  $3700,X
8BB8:9D 00 53 143      STA  $5300,X
8BBB:BD 00 3B 144      LDA  $3B00,X
8BBE:9D 00 57 145      STA  $5700,X
8BC1:BD 00 3F 146      LDA  $3F00,X
8BC4:9D 00 5B 147      STA  $5B00,X

8BC7:CA                149      DEX                ; ONE MORE ROW?
8BC8:E0 7F             150      CPX  #$7F           ; BYPASS UNUSED LOCATIONS?
8BCA:D0 02             151      BNE  NOFIX
8BCC:A2 77             152      LDX  #$77
8BCE:E0 FF             153      NOFIX CPX  $FF
8BD0:F0 03             154      BEQ  ROWDUN
8BD2:4C 0D 8B          155      JMP  NXODD
8BD5:A2 27             157      ROWDUN LDX  #$27      ; HANDLE SPECIAL MAPPINGS
8BD7:BD 80 20          158      ODDTHD LDA  $2080,X    ;      8 TO 7
8BDA:9D 00 5C          159      STA  $5C00,X
8BDD:BD 28 20          160      LDA  $2028,X    ;      64 TO 63
8BE0:9D 80 5F          161      STA  $5F80,X
8BE3:BD A8 20          162      LDA  $20A8,X    ;      72 TO 71
8BE6:9D 28 5C          163      STA  $5C28,X
8BE9:BD 50 20          164      LDA  $2050,X    ;      128 TO 127
8BEC:9D A8 5F          165      STA  $5FA8,X
8BEF:BD D0 20          166      LDA  $20D0,X    ;      136 TO 135

```

PROGRAM 7-1, CONT'D...

```

8BF2:9D 50 5C 167      STA $5C50,X
8BF5:A9 00 168      LDA #00      ; ERASE BOTTOM LINES
8BF7:9D D0 3F 169      STA $3FD0,X  ; OF PAGE ONE
8BFA:9D D0 5F 170      STA $5FD0,X  ; OF PAGE TWO
8BFD:CA 171      DEX
8BFE:10 D7 172      BPL ODDTHD

8C00:20 09 8D 174      JSR SYNC      ; OPTIONAL LOCK TO VBLANK
8C03:8E 55 C0 175      STX PAGE2    ; SWITCH TO PAGE TWO

8C06:          177 ; *** START REMAP BACK TO PAGE ONE ***

8C06:A2 F7 179      LDX #$F7      ; BYPASS UNMAPPED BITS
8C08:BD 00 44 180 GOEVN LDA $4400,X ; LINES 1-65-129-9-...
8C0B:9D 00 20 181      STA $2000,X
8C0E:BD 00 48 182      LDA $4800,X
8C11:9D 00 24 183      STA $2400,X
8C14:BD 00 4C 184      LDA $4C00,X
8C17:9D 00 28 185      STA $2800,X
8C1A:BD 00 50 186      LDA $5000,X
8C1D:9D 00 2C 187      STA $2C00,X
8C20:BD 00 54 188      LDA $5400,X
8C23:9D 00 30 189      STA $3000,X
8C26:BD 00 58 190      LDA $5800,X
8C29:9D 00 34 191      STA $3400,X
8C2C:BD 00 5C 192      LDA $5C00,X
8C2F:9D 00 38 193      STA $3800,X

8C32:BD 00 41 195      LDA $4100,X ; LINES 16-80-144-24-...
8C35:9D 80 3C 196      STA $3C80,X
8C38:BD 00 45 197      LDA $4500,X
8C3B:9D 00 21 198      STA $2100,X
8C3E:BD 00 49 199      LDA $4900,X
8C41:9D 00 25 200      STA $2500,X
8C44:BD 00 4D 201      LDA $4D00,X
8C47:9D 00 29 202      STA $2900,X
8C4A:BD 00 51 203      LDA $5100,X
8C4D:9D 00 2D 204      STA $2D00,X
8C50:BD 00 55 205      LDA $5500,X
8C53:9D 00 31 206      STA $3100,X
8C56:BD 00 59 207      LDA $5900,X
8C59:9D 00 35 208      STA $3500,X
8C5C:BD 00 5D 209      LDA $5D00,X
8C5F:9D 00 39 210      STA $3900,X

```

PROGRAM 7-1, CONT'D...

8C62:BD 00 42 212	LDA \$4200,X	; LINES 32-96-160-40-...
8C65:9D 80 3D 213	STA \$3D80,X	
8C68:BD 00 46 214	LDA \$4600,X	
8C6B:9D 00 22 215	STA \$2200,X	
8C6E:BD 00 4A 216	LDA \$4A00,X	
8C71:9D 00 26 217	STA \$2600,X	
8C74:BD 00 4E 218	LDA \$4E00,X	
8C77:9D 00 2A 219	STA \$2A00,X	
8C7A:BD 00 52 220	LDA \$5200,X	
8C7D:9D 00 2E 221	STA \$2E00,X	
8C80:BD 00 56 222	LDA \$5600,X	
8C83:9D 00 32 223	STA \$3200,X	
8C86:BD 00 5A 224	LDA \$5A00,X	
8C89:9D 00 36 225	STA \$3600,X	
8C8C:BD 00 5E 226	LDA \$5E00,X	
8C8F:9D 00 3A 227	STA \$3A00,X	
8C92:BD 00 43 229	LDA \$4300,X	; LINES 48-112-176-56-...
8C95:9D 80 3E 230	STA \$3E80,X	
8C98:BD 00 47 231	LDA \$4700,X	
8C9B:9D 00 23 232	STA \$2300,X	
8C9E:BD 00 4B 233	LDA \$4B00,X	
8CA1:9D 00 27 234	STA \$2700,X	
8CA4:BD 00 4F 235	LDA \$4F00,X	
8CA7:9D 00 2B 236	STA \$2B00,X	
8CAA:BD 00 53 237	LDA \$5300,X	
8CAD:9D 00 2F 238	STA \$2F00,X	
8CB0:BD 00 57 239	LDA \$5700,X	
8CB3:9D 00 33 240	STA \$3300,X	
8CB6:BD 00 5B 241	LDA \$5B00,X	
8CB9:9D 00 37 242	STA \$3700,X	
8CBC:BD 00 5F 243	LDA \$5F00,X	
8CBF:9D 00 3B 244	STA \$3B00,X	

PROGRAM 7-1, CONT'D...

```

8CC2:CA      246      DEX
8CC3:E0 7F    247      CPX  #$7F      ; BYPASS UNUSED LOCATIONS?
8CC5:D0 02    248      BNE  NOFLX
8CC7:A2 77    249      LDX  #$77
8CC9:E0 FF    250 NOFLX  CPX  #$FF
8CCB:F0 03    251      BEQ  ROWDUM
8CCD:4C 08 8C 252      JMP  GOEVN

8CD0:A2 27    254 ROWDUM LDX  #$27      ; HANDLE SPECIAL MAPPINGS
8CD2:BD 80 40 255 EVNTHD LDA  $4080,X  ;      8 TO 7
8CD5:9D 00 3C 256      STA  $3C00,X
8CD8:BD 28 40 257      LDA  $4028,X  ;      64 TO 63
8CDB:9D 80 3F 258      STA  $3F80,X
8CDE:BD A8 40 259      LDA  $40A8,X  ;      72 TO 71
8CE1:9D 28 3C 260      STA  $3C28,X
8CE4:BD 50 40 261      LDA  $4050,X  ;      128 TO 127
8CE7:9D A8 3F 262      STA  $3FA8,X
8CEA:BD D0 40 263      LDA  $40D0,X  ;      136 TO 135
8CED:9D 50 3C 264      STA  $3C50,X
8CF0:BD D0 5F 265      LDA  $5FD0,X  ; RECOPY BOTTOM LINE
8CF3:9D D0 3F 266      STA  $3FD0,X
8CF6:CA      267      DEX
8CF7:10 D9    268      BPL  EVNTHD

8CF9:      270 ;      **** START REMAP BACK TO ONE ****

8CF9:88      272      DEY      ; NEXT SCREEN PAIR?
8CFA:F0 03    273      BEQ  DONE
8CFC:4C 05 8B 274      JMP  GOODD
8CFF:20 09 8D 275 DONE  JSR  SYNC      ; OPTIONAL LOCK TO VBLANK
8D02:8C 54 C0 276      STY  PAGE1      ; SWITCH TO PAGE ONE
8D05:20 3F FF 277      JSR  IOREST      ; RESTORE ALL REGISTERS
8D08:60      278      RTS      ; AND RETURN

8D09:      280 ;      **** VSYNC SUBROUTINE ****

8D09:2C 60 C0 282 SYNC  BIT  VSYNC      ; TEST FOR VBLANK
8D0C:10 FB    283      BPL  SYNC      ; AND WAIT FOR VBLANK
8D0E:60      284 NOSYNC RTS

*** SUCCESSFUL ASSEMBLY: NO ERRORS

```

GENTLE SCROLL															
8AFF-	EA														
8B00-	20	4A	FF	A0	04	20	09	8D							
8B08-	8E	54	C0	A2	F7	BD	00	24							
8B10-	9D	00	40	BD	00	28	9D	00							
8B18-	44	BD	00	2C	9D	00	48	BD							
8B20-	00	30	9D	00	4C	BD	00	34							
8B28-	9D	00	50	BD	00	38	9D	00							
8B30-	54	BD	00	3C	9D	00	58	BD							
8B38-	00	21	9D	80	5C	BD	00	25							
8B40-	9D	00	41	BD	00	29	9D	00							
8B48-	45	BD	00	2D	9D	00	49	BD							
8B50-	00	31	9D	00	4D	BD	00	35							
8B58-	9D	00	51	BD	00	39	9D	00							
8B60-	55	BD	00	3D	9D	00	59	BD							
8B68-	00	22	9D	80	5D	BD	00	26							
8B70-	9D	00	42	BD	00	2A	9D	00							
8B78-	46	BD	00	2E	9D	00	4A	BD							
8B80-	00	32	9D	00	4E	BD	00	36							
8B88-	9D	00	52	BD	00	3A	9D	00							
8B90-	56	BD	00	3E	9D	00	5A	BD							
8B98-	00	23	9D	80	5E	BD	00	27							
8BA0-	9D	00	43	BD	00	2B	9D	00							
8BA8-	47	BD	00	2F	9D	00	4B	BD							
8BB0-	00	33	9D	00	4F	BD	00	37							
8BB8-	9D	00	53	BD	00	3B	9D	00							
8BC0-	57	BD	00	3F	9D	00	5B	CA							
8BC8-	E0	7F	D0	02	A2	77	E0	FF							
8BD0-	F0	03	4C	0D	8B	A2	27	BD							
8BD8-	80	20	9D	00	5C	BD	28	20							
8BE0-	9D	80	5F	BD	A8	20	9D	28							
8BE8-	5C	BD	50	20	9D	A8	5F	BD							
8BF0-	D0	20	9D	50	5C	A9	00	9D							
8BF8-	D0	3F	9D	D0	5F	CA	10	D7							
8C00-	20	09	8D	8E	55	C0	A2	F7							
8C08-	BD	00	44	9D	00	20	BD	00							
8C10-	48	9D	00	24	BD	00	4C	9D							
8C18-	00	28	BD	00	50	9D	00	2C							
8C20-	BD	00	54	9D	00	30	BD	00							
8C28-	5B	9D	00	34	BD	00	5C	9D							
8C30-	00	38	BD	00	41	9D	80	3C							
8C38-	BD	00	45	9D	00	21	BD	00							
8C40-	49	9D	00	25	BD	00	4D	9D							
8C48-	00	29	BD	00	51	9D	00	2D							
8C50-	BD	00	55	9D	00	31	BD	00							
8C58-	59	9D	00	35	BD	00	5D	9D							
8C60-	00	39	BD	00	42	9D	80	3D							
8C68-	BD	00	46	9D	00	22	BD	00							
8C70-	4A	9D	00	26	BD	00	4E	9D							
8C78-	00	2A	BD	00	52	9D	00	2E							
8C80-	BD	00	56	9D	00	32	BD	00							
8C88-	5A	9D	00	36	BD	00	5E	9D							
8C90-	00	3A	BD	00	43	9D	80	3E							
8C98-	BD	00	47	9D	00	23	BD	00							
8CA0-	4B	9D	00	27	BD	00	4F	9D							
8CA8-	00	2B	BD	00	53	9D	00	2F							
8CB0-	BD	00	57	9D	00	33	BD	00							
8CB8-	5B	9D	00	37	BD	00	5F	9D							
8CC0-	00	3B	CA	E0	7F	D0	02	A2							
8CC8-	77	E0	FF	F0	03	4C	08	8C							
8CD0-	A2	27	BD	80	40	9D	00	3C							
8CD8-	BD	28	40	9D	80	3F	BD	A8							
8CE0-	40	9D	28	3C	BD	50	40	9D							
8CE8-	A8	3F	BD	D0	40	9D	50	3C							
8CF0-	BD	D0	5F	9D	D0	3F	CA	10							
8CF8-	D9	88	F0	03	4C	05	8B	20							
8D00-	09	8D	8C	54	C0	20	3F	FF							
8D08-	60	26	60	C0	10	FB	60								

Fig. 7-8. Hex dump of GENTLE SCROLL.SET.

A final detail finishes the mapping from page One to page Two. We clear the bottom line of both page One and page Two. If you don't do this, the descenders of lower-case characters will elongate and look very strange.

When the mapping is completed, we jump to the optional SYNC subroutine and, then, flip the switch to page Two.

The whole remapping process is then repeated, only this time we display page Two and remap back to page One. A dot initially on page One gets mapped one dot higher onto page Two and, then, gets remapped two dots up from where it started back on page One.

Eight remappings are done in four groups of two each. We end up back on page One with all the characters moved up exactly one dot row.

USE HINTS

Normally, you will save your program as a GENTLE SCROLL.SET on a disk containing the HRCG and LOADHRCG programs. When asked for the number of alternate character sets, answer one more than what you intend to use, and enter the GENTLE SCROLL.SET as your last alternate character font to be loaded.

A ready-to-go copy of the machine-language object code (GENTLE SCROLL.SET) is provided on the companion disk to this volume. The Apple HRCG will automatically load the object code when it is entered by name as the highest alternate character set.

Should you want to copy this program "by hand," do the following:

1. Boot a DOS 3.3 disk that has room on it for the GENTLE SCROLL.SET.
2. Get into the monitor by doing a CALL -151 <cr>.
3. Enter the hex dump code of Fig. 7-8. To do this, type 8AFF : EA <cr>. Then, type : 20 4A FF A0 04 20 07 8D <cr>. Continue this process of entering a colon, followed by eight op code bytes and a <cr> till you have entered all the code.
4. Type 8AFF L <cr> and verify the code by comparing it against Program 7-1. Retype L <cr> as often as needed to go all the way through the program.
5. Restore DOS by using 3DOG <cr>.
6. Type BSAVE GENTLE SCROLL.SET, A\$8AFF, L\$210. This should save your program to disk.
7. To use your gentle scroll, do a BLOAD GENTLE SCROLL.SET <cr> as needed. If you are NOT using HRCG, note that (1) you must protect the space 8AFF-8DOC, (2) you must be displaying text on HIRES page One, and (3) a CALL 35583, a CALL -29953, or a JSR 8AFF will move everything up eight dots and, then, return to your main program.

To activate your gentle scroll set, you have to go into the HRCG and find the subroutine call to its own abrupt scroll program. In machine language, use \$9214: 00 8B to use the gentle scroll and use \$9214: 21 92 to use the abrupt scroll. Equivalent Applesoft locations appear in Program 7-1.

These locations assume that you are using the same version HRCG that I am on a 48K Apple microcomputer. The hooks may change with a change in HRCG version or a change of program. What you do is reach into the host program and find that program's own call to its own abrupt scroll and, then, substitute the gentle scroll subroutine's starting address when it is wanted. There are more details on this in Enhancement 3.

Note that the GENTLE SCROLL.SET is not relocatable as is. It will only work with a starting address of either \$8AFF or \$8B00. There are three absolute jumps needed that are beyond the range of a relative branch, and there are absolute calls to the SYNC subroutine.

Should you want to use the gentle scroll with some other HIRES character generator as host, just find the equivalent scroll hooks and use them. If you want to test the gentle scroll without a character-generator program, make sure you are displaying HIRES page One and do a JSR to \$8B00, a CALL 35583, or a CALL —29953. This should move everything up one character row for you and then return you to your host program.

Program 7-2 is an Applesoft demo called GENTLE SCROLL TESTER. You can use this one without any HIRES character generator. The program puts a lower-case message on the screen, and slowly scrolls it up for you. The process repeats over and over again until you press any key.

Once again, all that the GENTLE SCROLL.SET can do for you is move existing HIRES characters or graphics up the screen. It cannot put anything new on the screen. For best effects, you will have to use *HRCCG* or something similar to first enter your HIRES screen message.

If you do not have the field sync mod in place, change \$8D09 to \$60 in Program 7-1. Otherwise, your gentle scroll may hang.

And, again repeating, you will get the smoothest results if you have the hardware field sync modification of Enhancement 4 in place.

Naturally, the gentle scroll will only work when a HIRES page is being displayed. Should you drop back to a conventional LORES/text page, the gentle scroll will no longer work. A system RESET will sometimes drop you out of a HIRES character-generator program. To restore your gentle scroll, reactivate the HIRES display mode, and verify that the scroll hooks are still intact.

Two gotchas. If you are using Applesoft POKES to connect the GENTLE SCROLL.SET to HRCCG, make absolutely sure you do it at the *top* of a screen. If you try this at the bottom of the screen, the first POKE sets up a scrolling address that is only "one half" correct, and the next scroll attempt bombs the program. And, never enter text while a gentle scroll is taking place as keys will be ignored for the time it takes to complete the scrolling. You can beat this with an add-on key buffer, but it is simpler to arrange your programs so as to never call for or accept a key input until after the display is stable.

Like so

TWO USE RULES

1. If you are using Applesoft POKE commands to connect your GENTLE SCROLL.SET, do so only at the TOP of the screen!
2. Do not call for or allow any text input while the gentle scrolling is actually taking place.

If your scroll seems slightly erratic, check first to make sure that the hardware field sync mod is in place and that Program 7-1 is hooked into it. It also pays to have the fastest possible entry of characters on the screen, since the gentle scroll has to take time out to let the HRCCG, or whatever else you use, put each

PROGRAM 7-2
GENTLE SCROLL TESTER

LANGUAGE: APPLESOFT

NEEDS: GENTLE SCROLL SET
FIELD SYNC MOD
(OPTIONAL)

```
10 REM *****
12 REM *
14 REM * GENTLE SCROLL *
16 REM * TESTER *
18 REM *
20 REM * VERSION 1.0 *
22 REM * (9-28-81) *
23 REM *
24 REM * COPYRIGHT 1981 *
26 REM * BY DON LANCASTER *
28 REM * AND SYNERGETICS *
30 REM *
32 REM * ALL COMMERCIAL *
34 REM * RIGHTS RESERVED *
36 REM *
38 REM *****

50 REM THIS PROGRAM TESTS AND
52 REM DEMONSTRATES THE GENTLE
54 REM SCROLL OF ENHANCEMENT 7
56 REM FOUND IN ENHANCING YOUR
58 REM APPLE II, VOLUME I.

60 REM THE SUBROUTINE "GENTLE
62 REM SCROLL.SET" IS NEEDED.

64 REM THE FIELD SYNC MOD
66 REM OF ENHANCEMENT #4 IS
68 REM NOT NEEDED, BUT WILL
70 REM GIVE A MUCH SMOOTHER
72 REM SCROLLING ACTION.

80 REM FOR BEST EFFECTS,
82 REM USE THE GENTLE SCROLL
84 REM WITH A HIRES CHARACTER
86 REM GENERATOR SUCH AS
88 REM HRCG OR HIGHER TEXT
90 REM INSTEAD OF THIS PROGRAM.

100 TEXT : HOME : VTAB 20: PRINT
    "HELLO! I AM YOUR ...."
```

PROGRAM 7-2, CONT'D...

```

105 HIMEM: 35500: REM    PROTECT
    GENTLE SCROLL SET SPACE
110 PRINT "BLOAD GENTLE SCROLL.S
    ET": REM  CTRL D

200 HGR : POKE  - 16302,0: CALL
    - 1998: REM  FULL HIRES SCR
    EEN
210 HCOLOR= 3
220 H = 100:V = 184: REM  SET TE
    XT START

230 GOSUB 2000
235 FOR N = 1 TO 1000: NEXT N: CALL
    35583: REM  "GENTLE"
240 H = 100: GOSUB 2000
245 FOR N = 1 TO 1000: NEXT N: CALL
    35583: REM  "SCROLL"
250 H = 100: GOSUB 2000: REM  "TE
    STER"

255 FOR N = 1 TO 3000: NEXT N
260 FOR N = 1 TO 9: CALL 35583: NEXT
    N
265 FOR N = 1 TO 3000: NEXT N
267 IF PEEK (49152) > 127 THEN
    POKE 49168,0: PRINT : PRINT
    "RUN MENU"
268 REM  DELETE 267 IF AUTO MENU
    IS NOT IN USE.

270 IF PEEK (49152) > 127 THEN
    POKE 49168,0: TEXT : HOME :
    PRINT "PARTING IS SUCH SWEE
    T SORROW": VTAB 22: END
280 CLEAR : HIMEM: 35582: GOTO 2
    20: REM  GO ROUND AGAIN IF N
    O KEY HAS BEEN PRESSED

1000 REM  CHARACTERS C,E,G,L,N,O
    ,R,S,T FOLLOW IN ORDER

1010 HPLOT H + 4,V + 2 TO H + 1,
    V + 2: HPLOT H,V + 3 TO H,V +
    5: HPLOT H + 1,V + 6 TO H +
    4,V + 6: RETURN
1020 HPLOT H,V + 4 TO H + 4,V +
    4 TO H + 4,V + 3: HPLOT H +
    3,V + 2 TO H + 1,V + 2: HPLOT
    H,V + 3 TO H,V + 5: HPLOT H +
    1,V + 6 TO H + 4,V + 6: RETURN

```

PROGRAM 7-2, CONT'D...

```
1030 HPlot H + 1,V + 2 TO H + 3,
      V + 2: HPlot H,V + 3 TO H,V +
      4: HPlot H + 1,V + 5 TO H +
      4,V + 5: HPlot H + 4,V + 3 TO
      H + 4,V + 6: HPlot H + 1,V +
      7 TO H + 3,V + 7: RETURN
1040 HPlot H + 1,V TO H + 2,V TO
      H + 2,V + 6: HPlot H + 1,V +
      6 TO H + 3,V + 6: RETURN
1050 HPlot H,V + 6 TO H,V + 2 TO
      H + 3,V + 2: HPlot H + 4,V +
      3 TO H + 4,V + 6: RETURN
1060 HPlot H,V + 3 TO H,V + 5: HPlot
      H + 1,V + 6 TO H + 3,V + 6: HPlot
      H + 4,V + 5 TO H + 4,V + 3: HPlot
      H + 1,V + 2 TO H + 3,V + 2: RETURN
1070 HPlot H,V + 2 TO H,V + 6: HPlot
      H,V + 3 TO H + 1,V + 3: HPlot
      H + 2,V + 2 TO H + 4,V + 2: RETURN
1080 HPlot H + 4,V + 2 TO H + 1,
      V + 2: HPlot H,V + 3: HPlot
      H + 1,V + 4 TO H + 3,V + 4: HPlot
      H + 4,V + 5: HPlot H,V + 6 TO
      H + 4,V + 6: RETURN
1090 HPlot H,V + 2 TO H + 4,V +
      2: HPlot H + 1,V TO H + 1,V +
      5: HPlot H + 2,V + 6 TO H +
      3,V + 6: HPlot H + 4,V + 5: RETURN
1999 REM

2000 FOR N = 1 TO 6: GOSUB 3000:
      H = H + 8: NEXT N: RETURN : REM
      PUIT DOWN SIX LETTERS

3000 READ CHARACTER: ON CHARACTE
      R GOSUB 1010,1020,1030,1040,
      1050,1060,1070,1080,1090
3010 RETURN : REM FIND LETTER C
      ODES

9999 DATA 3,2,5,9,4,2,8,1,7,6,4,
      4,9,2,8,9,2,7: REM SAYS "GE
      NTLE/SCROLL/TESTER"
```

line of characters on the screen. If you make each character line the same length and put them on the screen with fast machine-language code, you will get the smoothest results.

Funny things may happen if you try to use GENTLE SCROLL.SET on a colored background; particularly, if it's an "illegal" color. To handle colored backgrounds, you will have to suitably modify the "clear both bottom lines" code into something that handles each color byte individually. One alternative is to simply copy the bottom line of page One to the bottom line of page Two. This gives you color but introduces a lower-case elongating descender bug.

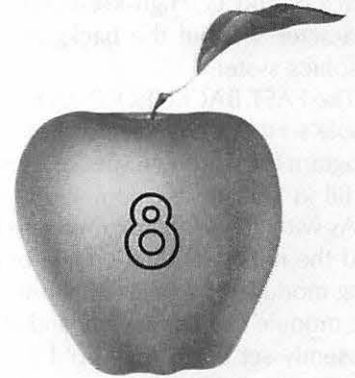
To try this color background alternative, replace GENTLE SCROLL.SET 8BF5: A9 00 9D 00 3F with 8BF5: BD D0 3F EA EA.

Once your gentle scroll is working, you should be able to think up all sorts of new possibilities. How would you handle the terrain flyby in a bomber mission or a road in a road race? How about dice, roulette wheels, or menu selectors that rotate? What about several things that are moving at once? Note that partial screen gentle scrolls can run much faster than whole screen ones. Note also that for graphics use, you might be able to jump two or more scan lines at a time, rather than just one 🍏

The programs GENTLE SCROLL.SET, GENTLE SCROLL.SET.SOURCE, and GENTLE SCROLL TESTER are included on the companion diskette to this volume.

All are fully copyable.

Enhancement



FAST BACKGROUND

A HIRES utility that gives you any of 191 solid background colors or 18,446,744,073,709,551,616 patterns. It runs seven times faster than you might expect and it is easily made glitchless.

FAST BACKGROUND

There is a convenient and useful HIRES screen clear subroutine that is available as part of your Apple's firmware. This code is provided both in the Applesoft ROMs and in the Programmer's Aide that comes with the Integer ROMs. The background clearing sub is easily reached from any language and is the "standard" way of clearing either HIRES screen.

It is also pitiful.

It is pitifully *slow* in that it takes over eight times as long to clear the screen as is necessary, resulting in a slow and ugly glitch during screen clearings. And, it is pitifully *weak* in that it only lets you clear to a paltry 8 of the 191 HIRES colors.

The FAST BACKGROUND is an incredibly fast machine-language module that will give you a right-now clear of HIRES to your choice of any 32 of the possible 191 color backgrounds or 18,446,744,073,709,551,616 color back-

ground patterns. The module is designed to load into the protected program space of HRCG (High-Resolution Character Generator) as the highest alternate character set, but the backgrounder can be used from any language or any graphics system.

The FAST BACKGROUNDER is also useful in showing us quite a bit about the Apple's HIRES color limits and capabilities. You can easily adapt parts of the program for your own special uses. You can also use the magic bit combinations to fill in your own colors in any way that you like.

As with any program, there are trade-offs. In exchange for the blinding speed and the mind-boggling choice of colors and patterns, we end up with a fairly long module (512 bytes) that only can do a full clear of HIRES page One. While the module can handle any and all of the HIRES patterns and colors, its files are presently set up to only hold 32 different colors or patterns at any one time. More patterns or colors are easily swapped back and forth to disk or from your controlling program. You can easily customize the backgrounder or use the ideas behind it to handle almost any clear of any size, shape, or page that you like. You can also easily modify this upcoming FAST BACKGROUNDER.SET program to handle either page Two or your choice of either HIRES page.

While the remaining visual glitch with this program is utterly negligible compared to the ugly transient that you get from the usual HIRES slow clear, we will also show you a way to get an absolutely glitchless and "invisible" clear to any color in the blink of an eye.

Let's see how the Backgrounder works

Lotsa dots

How much do you know about the HIRES capabilities of the Apple? Which one of the following is correct?

- () In HIRES, the Apple can put a single dot in any of 560 possible positions on any horizontal line.
- () In HIRES, the Apple can only put a maximum of 280 dots on any horizontal line.
- () In HIRES, the Apple can only put a maximum of 140 green dots on any horizontal line.
- () Under worst-case conditions, the Apple will only allow 40 HIRES color changes across the screen.

The answer, of course, is yes.

All of these statements are true. The Apple's horizontal resolution in HIRES is 40 dots, 140 dots, 280 dots, or 560 dots, depending on what you care to call a dot and how that dot has to relate to the others.

Let's see if we can't straighten this mess out some. Refer back to Fig. 7-5 of the previous enhancement to get us started.

As Fig. 7-5 showed us, there are forty bytes stored in memory for each horizontal line. Each byte is responsible for seven dots on the line. The leftmost byte is called the *base address*. The bytes continue in memory as base address +0, base address +1, base address +2, and so on, to base address +\$27, which is the fortieth and rightmost byte on the line. The actual HIRES addresses for both pages were listed for you in Table 7-1 of the previous enhancement.

As the sketch in Fig. 7-5 shows us, each HIRES byte, in turn, has eight bits. Seven of these bits are used to represent dots on one horizontal line. These dots are lit if the bit is a "one" and are unlit or black if the bit is a "zero."

The bottom seven bits in the byte map themselves *backwards* onto the screen. Thus, the least significant bit is the earliest and the leftmost, and the next-to-most significant seventh bit is the latest and, thus, the rightmost one to get plotted on the screen.

What about the eighth, or most significant bit? This bit acts as a shifter that either does nothing or else moves *all* of the other seven dots one-half of a dot to the left. This shifting is detailed in Figs. 8-1 and 8-2.

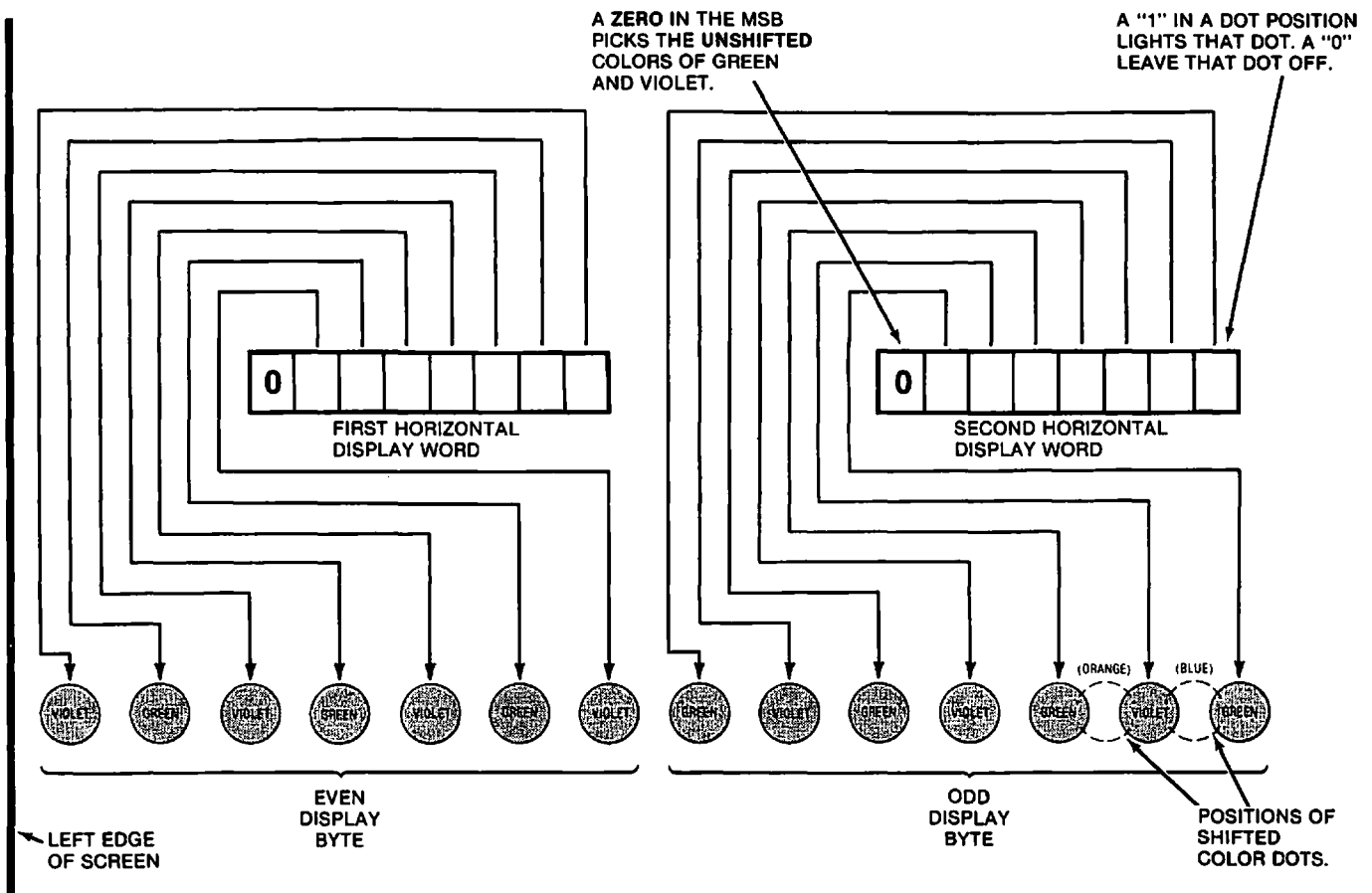


Fig. 8-1. How unshifted colors are mapped onto HIRES screen. Note that even and odd word color bit positions are different.

If the most significant bit is a zero, all the dots go where you would expect them to go. If the most significant bit is a one, all seven dots are all shifted together one-half dot to the right of their normal position.

There are two reasons why you might like to shift all seven dots. In a black and white HIRES display, a half-dot shift of a dot can appear to double the apparent resolution on a slanty line, giving you the illusion of a 560-dot horizontal resolution. Unfortunately, all seven dots in a byte must shift or unshift together, so this illusion fails if there is too much detail in the picture. The double-resolution illusion works best on simple large line figures that do not overlap.

Shifting is also useful to smooth out or "round" characters in a HIRES character set, or for other small and solid symbols where a half-step horizontal offset will even things out. Once again, all seven dots of a single byte must be shifted or unshifted at once.

The second reason why you would want to shift all seven dots is that it gives you new colors on the screen.

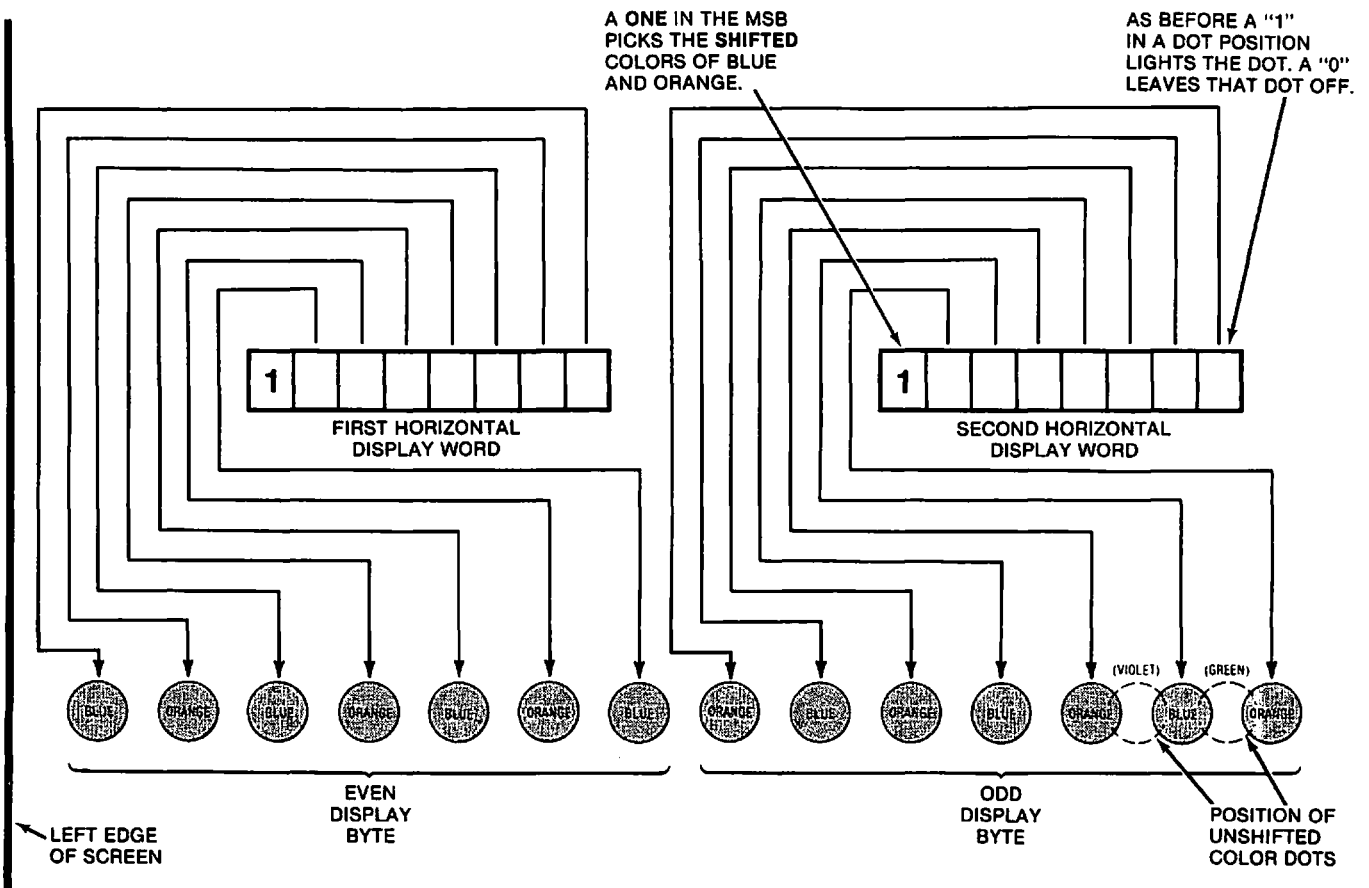


Fig. 8-2. How shifted colors are mapped onto HIRES screen. MSB of each display word shifts or unshifts seven dots at once.

Each dot position on a black and white set can either be lit or unlit. Lit gives you a white dot and unlit gives you a black dot. There are 280 possible dots on a horizontal line, equal to 40 bytes of 7 bits each. These dots can be placed in any of 560 possible positions, provided each group of 7 dots is shifted or not shifted as a block.

You will get the same black and white display on a color tv set if you cancel the color burst with the software-controlled color killer of Enhancement 2, or if you back all the color controls completely off.

Things get much more complicated if you want a color display. There is no way to produce a single white dot on a color tv set when the set is in its color mode. Each dot has to be a color since each dot consists of some red, blue, or green phosphor bars or dots on the screen.

One way to gain insight into how a color tv works in an Apple display is to assume that the exact *position* of the dot determines the *color* that will be displayed. If your dots are unshifted, you can assume dot-position zero will be violet and dot-position one will be green and dot-position two will be violet again, and so on across the screen.

If you have shifted your dots one-half dot, as we did in Fig. 8-2, you can assume that dot-position zero will be blue and dot-position one will be orange and dot-position two will be blue again, and so on across the screen. Your Apple can give you either green and violet dots inside a 7-dot byte, or else, it can give you orange and blue dots, again inside a 7-dot byte. You get the green and violet by unshifting the seven dots and you get the orange and blue by shifting the dots.

What actually goes on inside the tv is more complicated. Each dot pair represents one cycle of the 3.58-megahertz color reference provided by the Apple and used by the tv. The phase shift, or relative time delay, between this reference burst and the presence of a dot decides the color for that dot. Once the color is decided, the red, blue, and green phosphors are lit in the magic combination needed to get the right color.

But, we are mainly interested in results rather than how those results are obtained. Instead, simply assume that each dot position is a unique color.

Like so . . .

Dot position 0.0 violet
 Dot position 0.5 blue
 Dot position 1.0 green
 Dot position 1.5 orange

Dot position 2.0 violet
 Dot position 2.5 blue
 Dot position 3.0 green
 Dot position 3.5 orange

. . . and so on across the screen.

Now for the tricky part. The way you get a white dot or line on a color set is to light a *pair* of dots. Since green and violet are complementary colors, lighting both of them beside each other will appear to give you a double-wide white dot. Similarly, orange and blue are complementary colors. Lighting a pair of these, side-by-side, will also give you a white dot.

Thus, it takes *two* dots, side-by-side, to give you the illusion of a white dot on a color tv. What really happens is that two adjacent dots of complementary colors get lit and you end up with the illusion of white light.

All of which says that the color resolution of your Apple is only half as good as the black and white resolution. You can only put down 140 green dots on a horizontal line. Similarly, you can only put down 140 dots of any of the three other colors of violet, orange, or blue. You also can only put down 140 white dots, since a white dot really is a pair of adjacent color dots.

Now, that sounds really awful, but most of it is the fault of the color tv, owing to the subcarrier method used to extract color. You can beat all this by going to a direct red, blue, and green video, like we may do in a later enhancement. If you are willing to directly enter the color guns, and if you are willing to add a small and very fast RAM after your Apple's circuitry, there is virtually no limit to the resolution, color range, or grey scale that you can get from your Apple.

Right now though, there is also a further limitation, since each seven adjacent dots must be shifted or unshifted together. Thus, you cannot normally have a blue dot and a green dot right beside each other. Color changes are best left for *different* 7-dot bytes, rather than being done inside a single 7-dot byte. At worst, you could be limited to as few as forty color changes across your screen if you are not careful about which colors have to go together.

By the way, these are "nominal" colors. Your color tv settings can make a big difference in what you see or get.

So, we see that there are absolute color limits which are set by the way the Apple and your color tv interact. We must obey these rules, at least for now. But, we are free to play any games within these limits to create the illusion of more colors than you'd think possible. This gets tricky, but it really works good.

Let's sum all this up

APPLE COLORS AND RESOLUTION

1. In black and white, the Apple can put 280 dots or undots across the screen.

These dots or undots can go in any of 560 possible locations so long as 7 dots or undots in a byte are shifted at the same time.

2. In color, the Apple can only put down 140 dots of a given color on a horizontal line. The only dot colors are green, violet, orange, and blue.

Green and violet are done with unshifted 7-dot bytes. Orange and blue are done with shifted 7-dot bytes. You cannot mix shifted and unshifted dots inside a 7-dot byte.

3. Also in color, the Apple can only give you a black and white resolution of 140 dots since a complementary dot pair must be lit to get the illusion of white light.

The general idea is to use larger areas and many dots to trick the eye into seeing colors that are not there. This, of course, is the way all color printing works. Fortunately, the eye is much better at resolving detail than it is in resolving color, so we can get away with stunts like this. Let's see what we can get in the way of

More colors

A color tv only has three colors it can produce. These are red, blue, and green. But, it obviously plays games with combinations of these colors to give you a wide spectrum of colors; even some that do not exist elsewhere.

You can do the same thing with your Apple. One possibility is to flash different colors in the same position on alternate fields. We may look at this in a future enhancement. This method will give you some apparently individual lines and some dots of different colors, but it also tends to flicker and it has other limiting factors.

Instead, we will note that most colors used on your Apple will be used over a fairly large area. The days of stick figures and open lines on Apple HIRES are long since gone. Most colors will be presented over a wide area, rather than as a single line. Our Backgrounder will use the entire screen to create the illusion of having lots of HIRES colors. The same ideas are easily used to fill in colors inside any shape you like.

The key to more colors is tricking the eye. One trick we can pull works in the vertical direction. If you take a pair of adjacent horizontal lines, and make one blue and the other green, you will see aqua, particularly if there are many line pairs in use. By itself, use of line pairs of color should take the basic 6 colors and extend them to 21. This happens since $6 + 5 + 4 + 3 + 2 + 1 = 21$, the number of possible pairs of 6 colors available.

Most of these "new" colors formed by pairs of horizontal lines are very appealing and useful. A few are downright awful. Some look very good on a color set, while others stand out on a black and white display. A very careful choice of colors can give you the best of both worlds, with stunning colors that can still be resolved easily on a black and white display. This little detail can get very sticky if you are designing programs that can run on either type of tv set.

Can we gain still more colors? What about the horizontal direction? Suppose we mix a color, then black, a color, then black, and so on? Or white, then a color, white, then a color, and continue this. Or, even alternate pairs of white and black spaced colors one line above the other? What you end up with is lots more colors. Some of these will have a texture to them and others will have lots of individual dots, sort of like the colors in the Sunday comics. Many of these new colors will be very useful.

We will look at fourteen of these "new" colors. This brings up the total of solid or nearly solid colors per horizontal line to 20. Now, go to your alternating line pairs, and you end up with a total $20 + 19 + \dots + 2 + 1 = 210$ possible HIRES colors. Nineteen of these have ugly black stripes in them, but I count 191 that are genuinely useful HIRES colors so far. You might find more when you start to look at odd-ball bit combinations.

The theoretical number of different HIRES colors is much higher than 210, but there are many duplicate, ugly, or useless results along the way.

We will define our background colors with an 8-byte color cell as shown in Fig. 8-3. The cell is 4 bytes long by 2 bytes high. We have to be 4 bytes long since some of the color patterns will not repeat exactly until 28 dots. We end up with 2 bytes in height because we may use alternating line pairs for some colors.

Our cell is two scan lines deep, starting with an even scan line and ending with an odd scan line. We then are free to use our second line in any of several ways. For instance, we can simply repeat the upper line for the traditional colors. We can make this second line white to lighten the color. In theory, we can leave the second line black to darken the color, but this tends to be too striped for most uses. We can also use our second line to mix hues for us, perhaps combining blue and green to get aqua, and so on.

Finally, we can use our second scan line to mess with the texture of our screen. We can use it to minimize the texture of the "comic book" colors as we will see shortly. Also, we can use this second scan line to purposely enhance texture—say, to emphasize a pattern or a design.

Each 8-byte color cell will be controlled by an 8-entry color pattern file. The file values start at the top and work across. Thus, the first pattern is in the upper left, the fourth one is in the upper right, the fifth one is in the lower left, and, finally, the eighth one controls the bottom right seven dots in the cell.

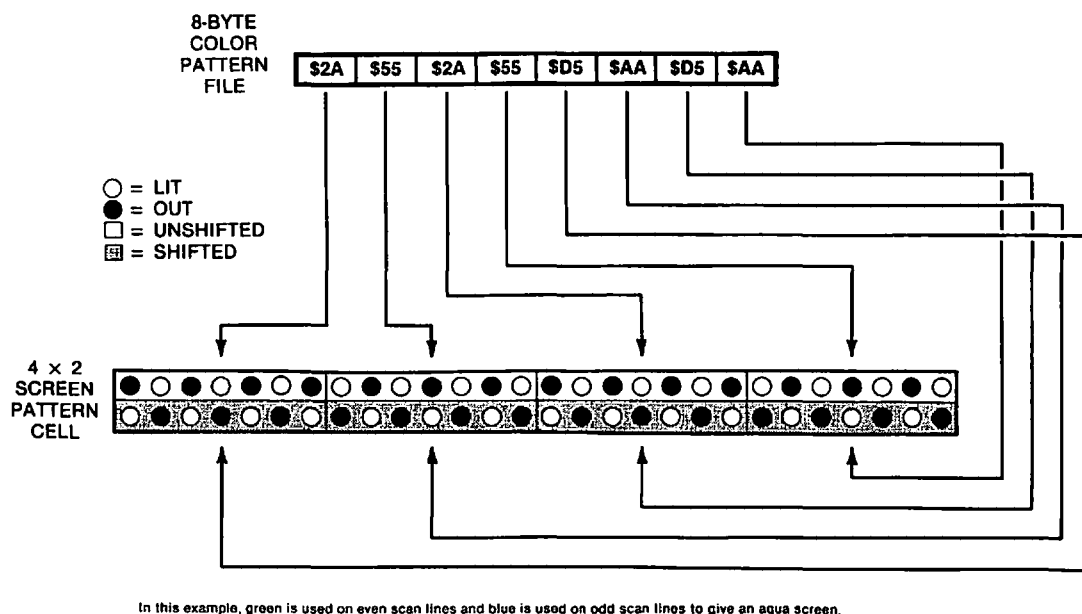


Fig. 8-3. Fast BACKGROUND.SET program uses an 8-byte color pattern file to map a 56-dot, 4x2-byte color block on the screen.

If our colors are going to appear to be continuous over the entire screen, we have to obey an important rule . . .

For continuous background color, the bit patterns must repeat EXACTLY when going across both 7-dot-byte boundaries and 8-byte cell boundaries.

What this says is that you have to be very careful in your choice of the ones and zeros in your individual bits and bytes. Otherwise, you will get lots of stripes across the screen and end up with a pattern instead of a color.

Now, if you want patterns, that's fine. Just stuff any old 64 bits into your cell (56 color dots and 8 half-dot shifts) and out comes a pattern. Some of these patterns are stunning. Some are ugly, others awful. Many are boring. Some can be used to, say, do a chain-link fence, or frost glass, or simulate a lace dress. Others are nice for curtain effects, used either in a window or as a full-screen stage.

There are more than several patterns available to you. Quite a few, in fact. Run through the mathematics and you'll find that there is a grand total of 186,446,744,073,709,551,616 possible patterns for your 8-byte cell.

This is so many patterns, in fact, that you might never be able to get through them all. So we will all share the work. Assume that there are only 125,000 readers of this book. A gruesomely conservative thought, but let's assume it anyway. Your share of the patterns will then only be a measly one and a half quadrillion patterns, give or take a few.

What I'd like you to do is this. Go through your share of the patterns one by one and when you find some really and truly outstanding ones, jot them down on the card that is in the back of this book and mail it in, and we'll publish the best patterns, textures, and colors that we get in a future volume of enhancements.

Pattern codes

Back to our colors. Seven is a nasty number. It is both odd and prime. Remember that we must get the dot patterns to repeat every byte and every cell, continuously, if we are to have a solid color rather than a pattern. This severely restricts the codes you can use. There are only *two* patterns that will repeat identically with every byte. There are four new patterns that will repeat identically with every second byte.

If we wait for a period of three bytes for our patterns to repeat, we pick up four more new and different patterns. Actually, these won't quite map into our upcoming 4-cell backgrounder space, and we will be violating our "it must fit exactly" rule. But, we will show you these three-byters just in case you want to play with them.

For maximum flexibility and the greatest possible choice of colors, we have to be willing to make four bytes, in a row, have different codings. We also have the option of stacking two different pairs of four bytes for even more color combinations.

The 4-byte patterns will give us ten new and useful combinations. Actually, there are many more 4-byte patterns possible, but these are either offset replicas of other 4-byte patterns, or else, they give the same color but a different texture. More details on this will be given shortly.

Summing things up . . .

NEW HIRES COLORS	
NO. OF BYTES	NO. OF NEW COLORS
One	2
Two	4
Three	4
Four	10
Total	20
Total number of combinations of pairs of 20 colors = $20 + 19 + 18 + 17 + \dots$, or $n(n+1)/2 = 210$.	
possible colors	210
– ugly colors	19
total	191
Total number of obvious and useful HIRES colors = 191.	

Someone else will most likely come up with some more subtle bit combinations that can lead to even more HIRES colors. But, no matter whose math you use, you will see that . . .

There are many more HIRES colors available on an Apple than there are LORES colors!

The big advantage of the 121 LORES colors of Enhancement 5 was that you got more hues, while with the HIRES combinations, you tend to get more shades

of fewer actual hues. Either route will lead to some mind-blowing color displays.

Let's look at these 1-, 2-, 3-, and 4-byte color combinations in detail. We will first look at combinations that only involve a single horizontal line.

There are only four possible bit patterns that leave all four bytes identical. Two of these are white, and two are black. Here's what they look like . . .

```
$00-00-00-00 ○○○○○○○○○○○○○○○○○○○○○○○○○○○○○ (white)
$80-80-80-80 ○○○○○○○○○○○○○○○○○○○○○○○○○○○○○ (white)*
$7F-7F-7F-7F ●●●●●●●●●●●●●●●●●●●●●●●●●●●● (black)
$FF-FF-FF-FF ●●●●●●●●●●●●●●●●●●●●●●●●●●●● (black)*
```

* = shifted color, ○ = lit dot

There are two black patterns and two white ones. Both blacks will look the same. The whites will be pretty much the same, although one may end up slightly "warmer" than the other. Usually, you use *unshifted* black and white with the upcoming green and violet, and use *shifted* black and white with orange and blue.

Note that there is no way to get any other continuous HIRES color by writing the same value to each and every byte. You can only get white and black this way. Any other values will give you stripes or patterns rather than solid colors.

If we go to pairs of horizontal bytes, we add these four solid colors . . .

```
$2A-55-2A-55 ●○○●○○●○○●○○●○○●○○●○○●○○●○○●○○ (green)
$AA-D5-AA-D5 ●○○●○○●○○●○○●○○●○○●○○●○○●○○●○○ (orange)*
$55-2A-55-2A ○●○○●○○●○○●○○●○○●○○●○○●○○●○○● (violet)
$D5-AA-D5-AA ○●○○●○○●○○●○○●○○●○○●○○●○○●○○● (blue)*
```

* = shifted color

These are the usual "eight" colors provided by Apple in its HIRES routines. You have four solid colors, two blacks, and two whites.

Note particularly how each second byte has to have a *different* bit pattern for the colors. The bit pattern has to be continuous over the screen. Since there are an odd number of bits per byte, this means that the 2-byte colors have to be stored as different values.

To make things more interesting, let's now look at four more new 3-byte patterns . . .

```
$49-25-12 ●○○●○○●○○●○○●○○●○○●○○●○○●○○●○○ (3/1)
$C9-A5-92 ●○○●○○●○○●○○●○○●○○●○○●○○●○○●○○ (3/2)*
$36-5B-6D ○●●○○●○○●○○●○○●○○●○○●○○●○○●○○ (3/3)
$B6-DB-ED ○●●○○●○○●○○●○○●○○●○○●○○●○○●○○ (3/4)*
```

* = shifted color

At first glance, you might think we would have more 3-color patterns than this. We already have used up the all-white and all-black patterns as 4-byte colors.

And, the other combinations are simply “phase shifts” or offsets of what we already have. For instance, the pattern \$25-12-49 gives us the same color as \$49-25-12, only with the actual dot pattern shifted one byte to the right. This offset shift will give us nothing new in the way of color, but might be useful to minimize any background pattern. We will see a good example of this with our 4-color patterns.

In general, the 3-color patterns are kind of bland greys and are not too exciting. Their use gets complicated by the need for us to handle things “by fours” in our upcoming fast background program. While we won’t be using these too much here, you might find the 3-byte colors useful additions to your bag of tricks.

Some of the 4-byte patterns are really neat. There are ten new and useful ones. We pick up four new pastels, two greys, and four dark colors. Let’s start with the pastels

\$11-22-44-08 ●○○○●○○○●○○○●○○○●○○○●○○○ (lime green)
 \$91-A2-C4-88 ●○○○●○○○●○○○●○○○●○○○●○○○ (sky blue)*
 \$22-44-08-11 ○●○○○●○○○●○○○●○○○●○○○●○○○ (lilac)
 \$A2-C4-88-91 ○●○○○●○○○●○○○●○○○●○○○●○○○ (beige)*
 * = shifted color

These four new “pastel” colors will also have four identical other ones, offset by two bytes. These add nothing new in the way of color, but give us a way to break up any background pattern.

For instance, we can do this for lime green . . .

```

●○○○●○○○●○○○●○○○
●○○○●○○○●○○○●○○○
●○○○●○○○●○○○●○○○
●○○○●○○○●○○○●○○○
●○○○●○○○●○○○●○○○

```

. . . here we used the same pattern on all four scan lines. But, if we alternate the regular and the offset pattern on alternate scan lines, note how the texture “breaks up”

```

●○○○●○○○●○○○●○○○
○○●○○○●○○○●○○○
●○○○●○○○●○○○●○○○
○○●○○○●○○○●○○○
●○○○●○○○●○○○●○○○

```

Both displays are lime green. But the bottom choice will give us a more uniform lime green, since the texture is broken up. This becomes most obvious with the upcoming dark colors.

We will shortly be listing some of these color patterns. A “normal” color pattern should usually be used on the *even* scan lines of any color area, while an “offset” color pattern should be used on the *odd* scan lines of any color area. You are, of course, free to do whatever you like for special effects. Sometimes, you might want to enhance texture rather than trying to minimize it.

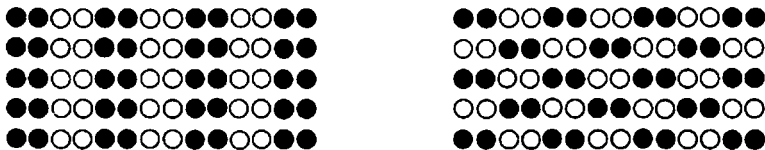
Here are our two new greys . . .

\$33-66-4C-19 ●●○○●●○○●●○○●●○○●●○○●●○○ (grey 1)

\$B3-E6-CC-A99 ●●○○●●○○●●○○●●○○●●○○●●○○●●○○ (grey 2)*

* = shifted color

Once again, there really are at least four greys—one normal and one offset by two bytes. Notice one more time how we can emphasize or reduce texture with the choice of regular or offset colors . . .



These are subtle differences to be sure, but they get important real quick when you try to build something that looks good both on a black and white and a color tv screen.

Our final four colors are dark ones, since they hit one color dot, miss the next dot of the same color, then hit the next one, and so on. Here is what they look like . . .

\$6E-5D-3B-77 ○●●●○●●●○●●●○●●●○●●●○●●●○●●●○ (purple)

\$DD-BB-F7-EE ○●●●○●●●○●●●○●●●○●●●○●●●○●●●○ (navy blue)*

\$5D-3B-77-6E ●○●●○●●●○●●●○●●●○●●●○●●●○●●●○ (forest green)

\$EE-DD-BB-F7 ●○●●○●●●○●●●○●●●○●●●○●●●○●●●○ (brown)*

* = shifted color

These tend to give a “comic book” or “Sunday funnies” effect at close range and are just what you need to add variety to a HIRES color palette.

You could probably go on to 5-byte colors, and 6-byte colors, and so on, but things would most likely end up so spread out that you no longer would see any apparently solid colors. You could also go to patterns of two lit, three off, and so on, as well. Chances are that most of these new patterns will either be obvious or else very near to what we already have. But—check them out anyway, since there are bound to be some useful surprises along the way.

If you do go into your own exotic patterns, be sure to remember our rule that the bit pattern on the screen must repeat *exactly* across each byte boundary and for each 8-byte, 4×2 cell.

Add everything useful up so far, and you get 20 one-line colors. We are now free to use our second scan line to add to our basic 20 colors. As we have seen, you can use this second line to duplicate colors, to offset bytes for minimum patterns, to mix hues together, to lighten, to darken, or to actually emphasize patterns. These new combinations give you at least 220 colors. This drops to 191 when you remove the 19 uglies that you get when you try to darken with a black line.

Table 8-1 shows my choice of color-file values for the thirty-two most useful colors and patterns. We start with Apple’s own six colors. Here the first and second scan lines are identical. Then we add fourteen of the 4-byte colors. With the 4-byte colors, we use the second scan line to offset and minimize any screen patterns.

Table 8-1. File Values for "Stock" Background Colors

NO.	COLOR	HEX FILE VALUES	DECIMAL FILE VALUES	SECOND LINE USE
0	black1	\$00-00-00-00-00-00-00	00-00-00-00-00-00-00	duplicate
1	green	\$2A-55-2A-55-2A-55-2A-55	42-85-42-85-42-85-42-85	duplicate
2	blue	\$D5-AA-D5-AA-D5-AA-D5-AA	213-170-213-170-213-170-213-170	duplicate
3	white1	\$7F-7F-7F-7F-7F-7F-7F-7F	127-127-127-127-127-127-127-127	duplicate
4	black2	\$80-80-80-80-80-80-80-80	128-128-128-128-128-128-128-128	duplicate
5	violet	\$55-2A-55-2A-55-2A-55-2A	85-42-85-42-85-42-85-42	duplicate
6	orange	\$AA-D5-AA-D5-AA-D5-AA-D5-AA	170-213-170-213-170-213-170-213	duplicate
7	white2	\$FF-FF-FF-FF-FF-FF-FF-FF	255-255-255-255-255-255-255-255	duplicate
8	purple	\$11-22-44-08-44-08-11-22	17-34-66-08-66-08-17-34	offset
9	navy	\$91-A2-C4-88-C4-88-91-A2	145-162-196-136-196-136-145-162	offset
10	forest	\$22-44-08-11-08-11-22-44	34-66-08-17-08-17-34-66	offset
11	brown	\$A2-C4-88-91-88-91-A2-C4	162-196-136-145-136-145-162-196	offset
12	grey1	\$33-66-4C-19-4C-19-33-66	51-102-76-25-76-25-51-102	offset
13	grey2	\$B3-E6-CC-99-CC-99-B3-E6	179-230-204-153-204-153-179-230	offset
14	lime	\$6E-5D-3B-77-3B-77-6E-5D	110-93-59-119-59-119-110-93	offset
15	lilac	\$5D-3B-77-6E-77-6E-5D-3B	93-59-119-110-119-110-93-59	offset
16	beige	\$EE-DD-BB-F7-BB-F7-EE-DD	238-221-187-247-187-247-238-221	offset
17	sky	\$DD-8B-F7-EE-F7-EE-DD-BB	221-187-247-238-247-238-221-187	offset
18	aqua	\$2A-55-2A-55-D5-AA-D5-AA	42-85-42-85-213-170-213-170	mix hues
19	steel	\$2A-55-2A-55-55-2A-55-2A	42-85-42-85-85-42-85-42	mix hues
20	yellow	\$2A-55-2A-55-AA-D5-AA-D5	42-85-42-85-170-213-170-213	mix hues
21	powder	\$D5-AA-D5-AA-55-AA-55-AA	213-170-213-170-85-42-85-42	mix hues
22	pink	\$D5-AA-D5-AA-AA-D5-AA-D5	213-170-213-170-170-213-170-213	mix hues
23	magenta	\$AA-D5-AA-D5-55-2A-55-2A	170-213-170-213-85-42-85-42	mix hues
24	olive	\$2A-55-2A-55-7F-7F-7F-7F	42-85-42-85-127-127-127-127	lighten
25	silver	\$D5-AA-D5-AA-FF-FF-FF-FF	213-170-213-170-255-255-255-255	lighten
26	salmon	\$AA-D5-AA-D5-FF-FF-FF-FF	170-213-170-213-255-255-255-255	lighten
27	mauve	\$55-2A-55-2A-7F-7F-7F-7F	85-42-85-42-127-127-127-127	lighten
28	drape1	\$C1-A3-A5-89-2C-08-CD-FE	193-163-165-137-44-8-205-254	pattern
29	drape2	\$9A-43-54-18-05-26-53-01	154-67-84-24-5-30-83-1	pattern
30	chain1	\$00-7F-00-7F-7F-00-7F-00	0-127-0-127-127-0-127-0	texture
31	chain2	\$0F-0F-0F-0F-78-78-78-78	15-15-15-15-120-120-120-120	texture

NOTE: Colors will vary with tv and its adjustments.

Next, in the file, are six mixed hue colors. Of these, aqua and magenta are particularly good looking. We do the mixing by putting down one color on one scan line and a second color on the second scan line. Following this are four colors in which the second scan line is white, giving us some additional pastels. Finally, we have four examples of patterns in which the scan lines are used to emphasize, rather than minimize the texture. Two stage "curtains" and two "chain" effects are shown.

Naturally, you are free to mix and match things anyway you want for your own color files. You can also use the same patterns to fill much smaller areas than the whole screen, although it will often take several scan lines and a few bytes of width to get a pleasing result.

Well, that's about all we need in the way of HIRES color theory. All you have to do to color any part of the screen 191 ways is to paint an 8-byte block of the screen with the magic bytes. If you need partial blocks, you simply continue the bit pattern up to the border of whatever it is you are coloring. Some colors will not be compatible with some borders, so you have to experiment to get the best overall results.

For patterns instead of colors, you do the same thing. The only difference is that you now have a mind-boggling choice of special effects at your disposal.

By the way, our color names are only rough guidelines. The actual colors you get will vary with the tv and its color settings.

Now all we need is a snappy program to rapidly clear the HIRES screen for us. Something like

FAST BACKGROUND.SET

Program 8-1 is a machine-language subroutine called FAST BACKGROUND.SET. While it is designed to fit in an alternate character slot in the HRCG of the *DOS Toolkit*, you can use it with any program in any language that needs a fast clear of HIRES1. Remember that the program must be located in a protected space.

Many of the ideas behind this program are the same as we used in the GENTLE SCROLL.SET of the previous enhancement. To give us the fastest possible screen erasure, we work only with the entire screen at once, use fast indexed store instructions, and share our loop overhead sixteen ways at once.

There are two main parts to the program. The operating code goes from \$8AFF to 8BFF, while a 32-value *color file* resides from \$8C00 to \$8CFF. To use the program, you poke or load your choice of color or pattern into location \$8B00, which is the same as an Applesoft POKE to 35584. This color value will be a number from 0 to 31. You then activate the fast background clear by jumping to a subroutine at \$8B01 or calling 35585.

Table 8-2 shows you the color-file values as they are now. Each color takes up 8 successive bytes in the file, giving us 32 total colors or patterns in the one-page-long file. You can change any or all of these color values to suit your own needs.

A flowchart of the FAST BACKGROUND.SET is shown in Fig. 8-4. We first calculate a file pointer by multiplying the color times eight. Then, we get the first color byte and use this byte in 1024 places, arranged as sixteen lines of 64 bytes per line. Note that we only map every *fourth* byte of only the *even* scan lines when we put this byte down.

We then get the second byte and put it down, again in 1024 locations. This is followed by the third byte and, finally, the fourth, which completes the even scan line mappings.

After that, we repeat the process on the odd scan lines, starting with the fifth byte in the selected color file and going on until we end up on the eighth and final value. When you are done, you have mapped 8 cells of 4×2 bytes into 1024 locations, or 8192 bytes total. Most of these bytes are on screen, while a few are off screen but not otherwise used.

Program 8-2 is an Applesoft demo that either shows you your choice of all 32 colors or patterns in order, or lets you enter and view your own patterns, or gives you a random-background show. It is menu driven.

For a 32-color show, the desired color gets POKED and the FAST BACKGROUND.SET is called. Then, a several second delay takes place, and you go on to the next color. Be sure to note that the screen clear takes place on the color *change*, and not on the long viewing delay. The screen clears or changes literally as fast as you can blink your eye.

To enter your own color or pattern, color-file location 0, or the eight bytes from \$8C00 to \$8C07, is borrowed, and the pattern is stuffed in here. This pattern is then displayed.

The random-color show does pretty much the same thing except the program throws its own eight random numbers into color-file Zero. Should you want to

PROGRAM 8-1 **FAST BACKGROUND SET**

LANGUAGE: APPLE ASSEMBLER

NEEDS: FIELD SYNC
MOD (OPTIONAL)

```

8AFF:      4 ; *****
8AFF:      5 ; *
8AFF:      6 ; *      FAST      *
8AFF:      7 ; *  BACKGROUND.SET *
8AFF:      8 ; *  ($8AFF.8CFF)  *
8AFF:      9 ; *
8AFF:     10 ; *  VERSION  1.1  *
8AFF:     11 ; *  (10-9-81 )  *
8AFF:     12 ; *
8AFF:     13 ; *  COPYRIGHT 1981 *
8AFF:     14 ; *  BY  DON LANCASTER *
8AFF:     15 ; *  AND SYNERGETICS *
8AFF:     16 ; *
8AFF:     17 ; *  ALL  COMMERCIAL *
8AFF:     18 ; *  RIGHTS RESERVED *
8AFF:     19 ; *
8AFF:     20 ; *****

8AFF:     22 ; THIS PROGRAM GIVES A FAST HIRES
8AFF:     23 ; CLEAR TO ANY OF THIRTY TWO
8AFF:     24 ; BACKGROUND COLORS OR PATTERNS

8AFF:     26 ; FROM HRCG --

8AFF:     28 ;      LOAD AS HIGHEST CHARACTER SET
8AFF:     29 ;      SET USER SUB A TO $8B01
8AFF:     30 ;      BY $9150: 01 8B
8AFF:     31 ;      STORE COLOR (0-$1F) AT $8B00
8AFF:     32 ;      CTRL-A CTRL-Y DOES IT

8AFF:     34 ; FROM MACHINE LANGUAGE --

8AFF:     36 ;      PUT COLOR ($0-$1F) IN $8B00
8AFF:     37 ;      JSR $8B01

8AFF:     39 ; FROM INTEGER BASIC --

8AFF:     41 ;      SET HIMEM > -29953
8AFF:     42 ;      POKE -29952, COLOR (0-31)
8AFF:     43 ;      CALL -29951

```

PROGRAM 8-1, CONT'D...

```
8AFF:          45 ;   FROM APPLESOFT --

8AFF:          47 ;           SET HIMEM < 35583
8AFF:          48 ;           POKE 35584, COLOR (0-31)
8AFF:          49 ;           CALL 35585

8AFF:          51 ;   THERE IS AN OPTIONAL "INVISIBLE"
8AFF:          52 ;   GLITCH ELIMINATOR BUILT INTO
8AFF:          53 ;   THIS PROGRAM.  TO USE IT, YOU
8AFF:          54 ;   MUST CLEAR THE LORES SCREEN TO
8AFF:          55 ;   GREY AND SHOULD HAVE THE FIELD
8AFF:          56 ;   SYNC MOD IN PLACE.

8AFF:          58 ;   TO ACTIVATE THE INVISIBLE SWITCHER --

8AFF:          60 ;           POKE 35586,159 OR $8B01: 9F

8AFF:          62 ;   TO TURN OFF THE INVISIBLE SWITCHER --

8AFF:          64 ;           POKE 35586,04 OR $8B02: 04


C057:          66 HIRES   EQU   $C057
C056:          67 LORES   EQU   $C056
C060:          68 SYNC    EQU   $C060
```

PROGRAM 8-1, CONT'D...

```

8AFF:          71 ;      ***** MAIN PROGRAM *****

8AFF:00          73          DFB $00          ; ADJUST HRCG SET START
8B00:00          74 COLOR  DFB $00          ; COLOR POKES HERE
8B01:4C 04 8B    75 PICK   JMP ERASE        ; OPTIONAL INVISIBLE LOCK
8B04:18          76 ERASE  CLC              ; FILE POINTER * 8
8B05:AD 00 8B    77          LDA COLOR      ; GET COLOR
8B08:0A          78          ASLA           ;
8B09:0A          79          ASLA           ;
8B0A:0A          80          ASLA           ;
8B0B:A8          81          TAY            ; SAVE COLOR FILE START
8B0C:A2 00       82          LDX #$00       ; AND PUT DOWN COLOR
8B0E:98          83 NXTBYT TYA             ; RESTORE POINTER
8B0F:48          84          PHA            ; AND SAVE ON STACK

8B10:B9 00 8C    86          LDA CFILE,Y    ; CHANGE EVEN SCAN LINES
8B13:A0 40       87          LDY #$40       ; FOR 64 TRIPS
8B15:9D 00 20    88 MAPEVN STA $2000,X    ;
8B18:9D 00 21    89          STA $2100,X    ;
8B1B:9D 00 22    90          STA $2200,X    ;
8B1E:9D 00 23    91          STA $2300,X    ;
8B21:9D 00 28    92          STA $2800,X    ;
8B24:9D 00 29    93          STA $2900,X    ;
8B27:9D 00 2A    94          STA $2A00,X    ;
8B2A:9D 00 2B    95          STA $2B00,X    ;
8B2D:9D 00 30    96          STA $3000,X    ;
8B30:9D 00 31    97          STA $3100,X    ;
8B33:9D 00 32    98          STA $3200,X    ;
8B36:9D 00 33    99          STA $3300,X    ;
8B39:9D 00 38    100         STA $3800,X    ;
8B3C:9D 00 39    101         STA $3900,X    ;
8B3F:9D 00 3A    102         STA $3A00,X    ;
8B42:9D 00 3B    103         STA $3B00,X    ;
8B45:CA          104         DEX            ; GO FOUR BLOCKS TO LEFT
8B46:CA          105         DEX            ;
8B47:CA          106         DEX            ;
8B48:CA          107         DEX            ; AND REPEAT
8B49:88          108         DEY            ; DONE WITH 64 BYTES?
8B4A:D0 C9       109         BNE MAPEVN     ;
8B4C:E8          110         INX            ; FOR NEXT COLOR BYTE
8B4D:68          111         PLA            ; GET BYTE COUNTER
8B4E:69 01       112         ADC #$01      ; AND INCREMENT
8B50:A8          113         TAY            ; SAVE NEXT COLOR FILE
                                         ; POINTER
8B51:29 03       114         AND #$03      ; CHECK FOR FOURTH TRIP
8B53:D0 B9       115         BNE NXTBYT    ; REPEAT FOR NEXT BYTE

8B55:A2 00       117         LDX #$00      ; AND PUT DOWN COLOR
8B57:98          118 NXBYTE TYA            ; RESTORE POINTER
8B58:48          119         PHA            ; AND SAVE ON STACK
8B59:B9 00 8C    120         LDA CFILE,Y   ; CHANGE ODD SCAN LINES
8B5C:A0 40       121         LDY #$40      ; FOR 64 TRIPS

```

PROGRAM 8-1, CONT'D...

```

8B5E:9D 00 24 122 MAPODD STA $2400,X ;
8B61:9D 00 25 123 STA $2500,X ;
8B64:9D 00 26 124 STA $2600,X ;
8B67:9D 00 27 125 STA $2700,X ;
8B6A:9D 00 2C 126 STA $2C00,X ;
8B6D:9D 00 2D 127 STA $2D00,X ;
8B70:9D 00 2E 128 STA $2E00,X ;
8B73:9D 00 2F 129 STA $2F00,X ;
8B76:9D 00 34 130 STA $3400,X ;
8B79:9D 00 35 131 STA $3500,X ;
8B7C:9D 00 36 132 STA $3600,X ;
8B7F:9D 00 37 133 STA $3700,X ;
8B82:9D 00 3C 134 STA $3C00,X ;
8B85:9D 00 3D 135 STA $3D00,X ;
8B88:9D 00 3E 136 STA $3E00,X ;
8B8B:9D 00 3F 137 STA $3F00,X ;
8B8E:CA 138 DEX ; GO FOUR BLOCKS TO LEFT
8B8F:CA 139 DEX ;
8B90:CA 140 DEX ;
8B91:CA 141 DEX ; AND REPEAT
8B92:88 142 DEY ; ONE LESS TRIP
8B93:D0 C9 143 BNE MAPODD ;
8B95:E8 144 INX ; FOR NEXT COLOR BYTE
8B96:68 145 PLA ; GET BYTE COUNTER
8B97:69 01 146 ADC #$01 ; AND INCREMENT
8B99:A8 147 TAY ; SAVE COLOR FILE LOCATION
8B9A:29 03 148 AND #$03 ; FOURTH TRIP?
8B9C:D0 B9 149 BNE NXBYTE ; REPEAT FOR NEXT BYTE

8B9E:60 151 RTS ; EXIT WHEN DONE

8B9F: 154 ; *** INVISIBLE SWITCHER ***

8B9F:2C 60 C0 156 TEST1 BIT SYNC ; LOOK FOR VBLANK
8BA2:10 FB 157 BPL TEST1 ;
8BA4:2C 56 C0 158 BIT LORES ; SWITCH TO LORES GREY
8BA7:20 04 8B 159 JSR ERASE ; DO FAST HIRES CLEAR
8BAA:2C 60 C0 160 TEST2 BIT SYNC ; FIND ANOTHER VBLANK
8BAD:10 FB 161 BPL TEST2 ;
8BAF:2C 57 C0 162 BIT HIRES ; SWITCH BACK TO HIRES
8BB2:60 163 RTS ; AND EXIT

```


PROGRAM 8-1, CONT'D...

```

8BB3:          165 ;      ***** COLOR PATTERN FILE *****

8C00:          167      ORG  COLOR+256

8C00:00 00 00 169 CFILE  DFB  $00,$00,$00,$00,$00,$00,$00,$00
8C03:00 00 00
8C06:00 00
8C08:2A 55 2A 170 PAT1  DFB  $2A,$55,$2A,$55,$2A,$55,$2A,$55
8C0B:55 2A 55
8C0E:2A 55
8C10:D5 AA D5 171 PAT2  DFB  $D5,$AA,$D5,$AA,$D5,$AA,$D5,$AA
8C13:AA D5 AA
8C16:D5 AA
8C18:7F 7F 7F 172 PAT3  DFB  $7F,$7F,$7F,$7F,$7F,$7F,$7F,$7F
8C1B:7F 7F 7F
8C1E:7F 7F
8C20:80 80 80 173 PAT4  DFB  $80,$80,$80,$80,$80,$80,$80,$80
8C23:80 80 80
8C26:80 80
8C28:55 2A 55 174 PAT5  DFB  $55,$2A,$55,$2A,$55,$2A,$55,$2A
8C2B:2A 55 2A
8C2E:55 2A
8C30:AA D5 AA 175 PAT6  DFB  $AA,$D5,$AA,$D5,$AA,$D5,$AA,$D5
8C33:D5 AA D5
8C36:AA D5
8C38:FF FF FF 176 PAT7  DFB  $FF,$FF,$FF,$FF,$FF,$FF,$FF,$FF
8C3B:FF FF FF
8C3E:FF FF
8C40:11 22 44 177 PAT8  DFB  $11,$22,$44,$08,$44,$08,$11,$22
8C43:08 44 08
8C46:11 22
8C48:91 A2 C4 178 PAT9  DFB  $91,$A2,$C4,$88,$C4,$88,$91,$A2
8C4B:88 C4 88
8C4E:91 A2
8C50:22 44 08 179 PAT10 DFB  $22,$44,$08,$11,$08,$11,$22,$44
8C53:11 08 11
8C56:22 44
8C58:A2 C4 88 180 PAT11 DFB  $A2,$C4,$88,$91,$88,$91,$A2,$C4
8C5B:91 88 91
8C5E:A2 C4
8C60:33 66 4C 181 PAT12 DFB  $33,$66,$4C,$19,$4C,$19,$33,$66
8C63:19 4C 19
8C66:33 66
8C68:B3 E6 CC 182 PAT13 DFB  $B3,$E6,$CC,$99,$CC,$99,$B3,$E6
8C6B:99 CC 99
8C6E:B3 E6
8C70:6E 5D 3B 183 PAT14 DFB  $6E,$5D,$3B,$77,$3B,$77,$6E,$5D
8C73:77 3B 77
8C76:6E 5D
8C78:5D 3B 77 184 PAT15 DFB  $5D,$3B,$77,$6E,$77,$6E,$5D,$3B
8C7B:6E 77 6E
8C7E:5D 3B

```

PROGRAM 8-1, CONT'D...

```

8C80:EE DD BB 186 PAT16 DFB $EE,$DD,$BB,$F7,$BB,$F7,$EE,$DD
8C83:F7 BB F7
8C86:EE DD
8C88:DD BB F7 187 PAT17 DFB $DD,$BB,$F7,$EE,$F7,$EE,$DD,$BB
8C8B:EE F7 EE
8C8E:DD BB
8C90:2A 55 2A 188 PAT18 DFB $2A,$55,$2A,$55,$D5,$AA,$D5,$AA
8C93:55 D5 AA
8C96:D5 AA
8C98:2A 55 2A 189 PAT19 DFB $2A,$55,$2A,$55,$55,$2A,$55,$2A
8C9B:55 55 2A
8C9E:55 2A
8CA0:2A 55 2A 190 PAT20 DFB $2A,$55,$2A,$55,$AA,$D5,$AA,$D5
8CA3:55 AA D5
8CA6:AA D5
8CA8:D5 AA D5 191 PAT21 DFB $D5,$AA,$D5,$AA,$55,$2A,$55,$2A
8CAB:AA 55 2A
8CAE:55 2A
8CB0:D5 AA D5 192 PAT22 DFB $D5,$AA,$D5,$AA,$AA,$D5,$AA,$D5
8CB3:AA AA D5
8CB6:AA D5
8CB8:AA D5 AA 193 PAT23 DFB $AA,$D5,$AA,$D5,$55,$2A,$55,$2A
8CBB:D5 55 2A
8CBE:55 2A
8CC0:2A 55 2A 194 PAT24 DFB $2A,$55,$2A,$55,$7F,$7F,$7F,$7F
8CC3:55 7F 7F
8CC6:7F 7F
8CC8:D5 AA D5 195 PAT25 DFB $D5,$AA,$D5,$AA,$FF,$FF,$FF,$FF
8CCB:AA FF FF
8CCE:FF FF
8CD0:AA D5 AA 196 PAT26 DFB $AA,$D5,$AA,$D5,$FF,$FF,$FF,$FF
8CD3:D5 FF FF
8CD6:FF FF
8CD8:55 2A 55 197 PAT27 DFB $55,$2A,$55,$2A,$7F,$7F,$7F,$7F
8CDB:2A 7F 7F
8CDE:7F 7F
8CE0:C1 A3 A5 198 PAT28 DFB $C1,$A3,$A5,$89,$2C,$08,$CD,$FE
8CE3:89 2C 08
8CE6:CD FE
8CE8:9A 43 54 199 PAT29 DFB $9A,$43,$54,$18,$05,$26,$53,$01
8CEB:18 05 26
8CEE:53 01
8CF0:00 7F 00 200 PAT30 DFB $00,$7F,$00,$7F,$7F,$00,$7F,$00
8CF3:7F 7F 00
8CF6:7F 00
8CF8:0F 0F 0F 201 PAT31 DFB $0F,$0F,$0F,$0F,$F8,$F8,$F8,$F8
8CFB:0F F8 F8
8CFE:F8 F8

```

*** SUCCESSFUL ASSEMBLY: NO ERRORS

Table 8-2. Color-File Locations Used in FAST BACKGROUND.SET

PATTERN	HEX LOCATION	DECIMAL LOCATION
0	\$8C00	35840
1	\$8C08	35848
2	\$8C10	35856
3	\$8C18	35864
4	\$8C20	35872
5	\$8C28	35880
6	\$8C30	35888
7	\$8C38	35896
8	\$8C40	35904
9	\$8C48	35912
10	\$8C50	35920
11	\$8C58	35928
12	\$8C60	35936
13	\$8C68	35944
14	\$8C70	35952
15	\$8C78	35960
16	\$8C80	35968
17	\$8C88	35976
18	\$8C90	35984
19	\$8C98	35992
20	\$8CA0	36000
21	\$8CA8	36008
22	\$8CB0	36016
23	\$8CB8	36024
24	\$8CC0	36032
25	\$8CC8	36040
26	\$8CD0	36048
27	\$8CD8	36056
28	\$8CE0	36064
29	\$8CE8	36072
30	\$8CF0	36080
31	\$8CF8	36088

In hex, LOCATION = \$8C00 + \$PATTERN *8.

In decimal, LOCATION = 35840 + PATTERN *8.

hold these values for later use, stop the program with a "Control C" at that point. To resume, type "RUN" as usual.

By the way, all the programs in this book have been modified to automatically exit you to the AUTO MENU program on the demo disk, so you can continuously run things without lots of extra keystrokes. Be sure to delete the "RUN MENU" exit lines if you do not want this feature in any of your programs.

ADD-ONS AND MODIFICATIONS

You can relocate the FAST BACKGROUND.SET anywhere you like, but notice that the file values will change. It is also important to start the color file on an exact page boundary. The space you pick must be protected from use by any other program. Usually, an Applesoft HIMEM command will do this for you either as a direct command or early in a program.

Naturally, you cannot see a fast background HIRES clear if you are in the text or LORES screen modes. To actually see the fast clear, you must be in HIRES1 when you do the erasure.

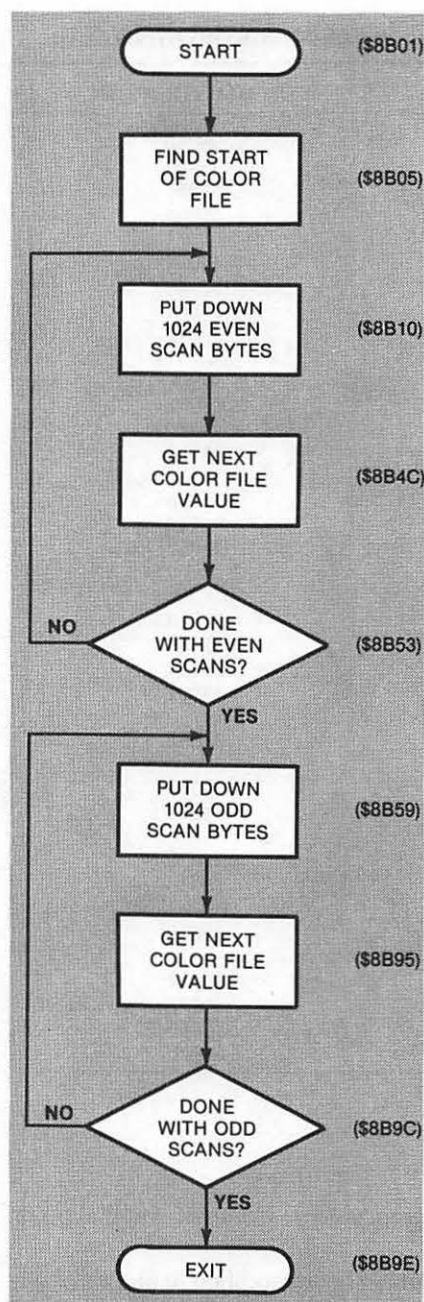


Fig. 8-4. Flowchart of FAST BACKGROUND.SET program.

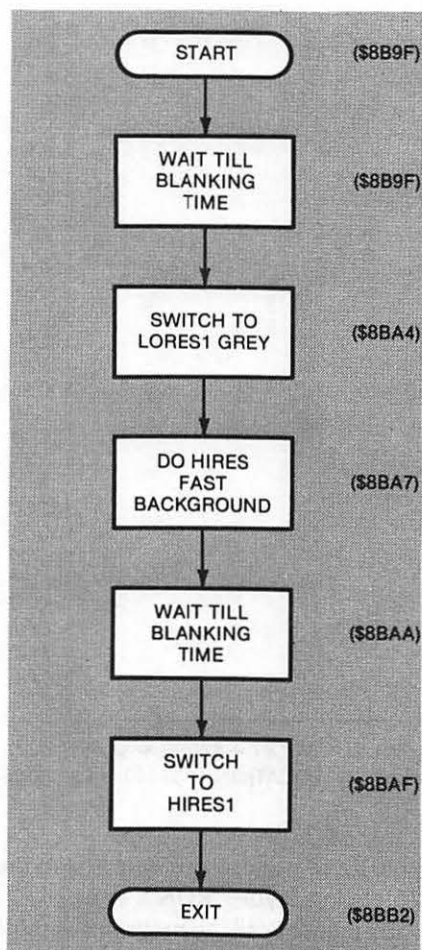


Fig. 8-5. Flowchart of optional "Invisible Switcher."

There are several different ways you can pick up a clear of HIRES page Two. To *only* clear HIRES2, just replace all the "2XXX" addresses with "4XXX" and all the "3XXX" addresses with "5XXX" ones. You can also duplicate the entire program and put it above \$8D00, sharing a common color file at \$8C00, with calls to the lower part giving you a HIRES1 clear and calls to the upper part giving you a HIRES2 clear.

Finally, if saving code space is important, you can also rework the MAPEVN and the MAPODD stores into subroutines. Jump to the "2XXX" and "3XXX" subs for a page One clear and to the "4XXX" and "5XXX" subs for a page Two

PROGRAM 8-2 FAST BACKGROUND DEMO

LANGUAGE: APPLESOFT

NEEDS: FAST BACKGROUND SET

```

10 REM *****
12 REM *
14 REM * FAST BACKGROUND *
16 REM * DEMO *
18 REM *
20 REM * VERSION 1.0 *
22 REM * (10-6-81) *
23 REM *
24 REM * COPYRIGHT 1981 *
26 REM * BY DON LANCASTER *
28 REM * AND SYNERGETICS *
30 REM *
32 REM * ALL COMMERCIAL *
34 REM * RIGHTS RESERVED *
36 REM *
38 REM *****

50 REM THIS PROGRAM TESTS AND
52 REM DEMONSTRATES THE FAST
54 REM BACKGROUNDER FOUND IN
56 REM ENHANCEMENT EIGHT OF
58 REM ENHANCING YOUR APPLE II
60 REM VOLUME I.

70 REM THE PROGRAM "FAST
72 REM BACKGROUND.SET" IS
74 REM IS ALSO NEEDED ON
76 REM THE SAME DISK.

80 REM THE FIELD SYNC MOD
82 REM OF ENHANCEMENT #4 IS
84 REM NOT NEEDED, BUT WILL
86 REM GIVE BETTER RESULTS
88 REM IF INSTALLED.

100 TEXT : HOME : CLEAR : PRINT
    : HTAB 08: PRINT "HIRES FAST
    BACKGROUND DEMO"
105 HIMEM: 35583: REM PROTECT F
    AST BACKGROUND.SET SPACE

110 PRINT ".....
    .....": PRINT
    : PRINT
115 PRINT "BLOAD FAST BACKGROUND
    .SET": REM CTRL D

```

PROGRAM 8-2, CONT'D...

```

117 POKE 35584,0: CALL 35585: REM
    HIDE THE HGR GLITCH
120 HTAB 6: PRINT "SHALL I .....
    ": PRINT : PRINT : PRINT
130 HTAB 9: PRINT "(S) SHOW STOC
    K COLORS": PRINT
140 HTAB 9: PRINT "(E) ENTER A N
    EW PATTERN": PRINT
150 HTAB 9: PRINT "(R) CREATE RA
    NDOM PATTERNS": PRINT
160 HTAB 9: PRINT "(Q) QUIT"
170 PRINT : PRINT : HTAB 18: PRINT
    "< >";: HTAB 19: REM PRINT
    MENU

200 GET A$: IF A$ = "S" THEN GOTO
    1000
210 IF A$ = "E" THEN GOTO 2000
220 IF A$ = "R" THEN GOTO 3000
230 IF A$ = "Q" THEN PRINT : PRINT
    "RUN MENU": REM EXIT
235 PRINT "": REM DING-DING-DIN
    G
240 HTAB 19: VTAB 19: GOTO 200: REM
    PICK MENU VALUE OR KEEP TR
    YING

1000 GOSUB 4000: HGR : REM GREY
    LORES FOR ERASE TIME
1010 VTAB 24: HTAB 5: PRINT "HIT
    ANY KEY TO RETURN TO MENU";

1015 VTAB 22: HTAB 12: PRINT "HI
    RES COLOR #";
1030 FOR C = 0 TO 31: POKE 35584
    ,C: HTAB 25: PRINT C;: CALL
    35585
1040 FOR N = 1 TO 2000: NEXT N: REM
    DISPLAY TIME
1050 IF PEEK ( - 16384) > 128 THEN
    POKE - 16368,0: GOTO 100
1060 FOR N = 1 TO 9: A = PEEK ( -
    16336): NEXT N: NEXT C: HTAB
    25: PRINT " ";: GOTO 1030
2000 HGR : PRINT
2005 VTAB 21
2010 PRINT "ENTER DECIMAL PATTEN
    N VALUES --": PRINT : PRINT

```

PROGRAM 8-2, CONT'D...

```

2020  FOR N = 1 TO 8
2030  VTAB 23: HTAB 1: INPUT " ";
      A$(N):A(N) = VAL (A$(N))
2040  VTAB 24: HTAB (N * 4 + 2): PRINT
      A(N);
2050  VTAB 23: HTAB 1: PRINT "
      ";: HTAB 1: NEXT N
2060  FOR N = 1 TO 8: POKE 35839 +
      N,A(N): NEXT N
2065  FOR N = 1 TO 9:A = PEEK ( -
      16336): NEXT N
2070  POKE 35584,0: CALL 35585
2080  PRINT : VTAB 24: PRINT
2085  PRINT : PRINT "<SPACE> NEW
      PATTERN      <M> MENU      < >";

2090  HTAB 37: 'GET A$
2100  IF A$ = " " THEN PRINT : PRINT
      : GOTO 2010
2110  IF A$ = "M" THEN GOTO 100
2120  GOTO 2090
3000  HGR : VTAB 24: HTAB 5: PRINT
      "HIT ANY KEY TO RETURN TO ME
      NU";
3010  VTAB 22: HTAB 1: PRINT "

      ";
3020  HTAB 5: FOR N = 1 TO 8:A(N)
      = INT (256 * RND (1)): PRINT
      A(N);: PRINT " ";: POKE 3584
      0 + N,A(N): NEXT N
3030  POKE 35584,0: CALL 35585: REM
      ENTER PATTERN ON SCREEN
3035  FOR N = 1 TO 9:A = PEEK ( -
      16336): NEXT N
3040  FOR N = 1 TO 3000: NEXT N: REM
      DISPLAY TIME
3050  IF PEEK ( - 16384) > 128 THEN
      POKE - 16368,0: GOTO 100
3060  GOTO 3010
4000  GR : COLOR= 5: FOR V = 0 TO
      39: HLIN 0,39 AT V: NEXT V: REM
      GREY LORES
4010  POKE 35586,159: REM INVISI
      BLE SWITCH ON
4020  RETURN
9999  END

```

clear. Note that you can fake an indirect subroutine jump with a subroutine jump followed by a jump indirect command.

Since there are lots of page Two options, we've purposely left them off the program to keep things simple. Add things on in any way that you like.

If you watch the background color demo, with line 4010 omitted, you will notice a faint "boxwork" glitch as you change colors. This glitch is utterly insignificant when compared to the royal mess you got with the old HIRES clear, but it is there.

There is an add-on to the FAST BACKGROUND.SET that totally eliminates this tiny remaining glitch for those of you who are program perfectionists. For this to work, you must have the field sync mod of Enhancement 4 in place and be willing to clear your text screen to LORES grey or some other suitable color.

Fig. 8-5 shows us the flowchart of the "Invisible Switcher." What this does is wait till a vertical blanking time, switches you to a perfect grey screen, does a new fast HIRES clear with the Backgrounder, waits for another blanking time, and, then, returns you to HIRES. Your eye sees only an instantaneous fade from one color through grey to another color, totally glitch free.

If you do not have the field sync mod in place, delete or bypass line 4010 of Program 8-2. Otherwise, the program may hang.

To use the Invisible Switcher, just poke 35586,159 or enter \$8B01 : 9F. Note that this invisible switcher is not for everyone, and it may in fact do more harm than good if you are always returning to the same background color. Nevertheless, it's there for those of you who are antiglitch purists. The Invisible Switcher is included in the demo of Program 8-2.

CONTENTS OF COMPANION PARTS KIT

- 1 each 74LS151 1/8 Selector IC.
- 2 each 74LS02 Quad NOR Gate IC.
- 2 each Tv clothespin connector.
- 4 each Phono jack, upright pc type.
- 4 each Spade lug, crimp style.
- 1 each 4.7K resistor, 1/4 watt.
- 3 each 16-pin DIP socket, premium machined-pin style.
- 1 each 14-pin DIP socket, premium machined-pin style.
- 1 each machined-pin socket contact.
- 6 inches No. 22 stranded wire, red insulation.
- 24 inches No. 24 solid wire, blue insulation.
- 6 inches No. 24 solid wire, red insulation.
- 12 inches Electronic solder.

The parts kit costs \$11.95 plus shipping and may be ordered using the attached card or directly from . . .

PAIA Electronics
Box 14359
1020 West Wilshire
Oklahoma City, OK 73114
Tel: 405-842-5480

**CONTENTS OF COMPANION
DISKETTE**

Auto Menu
Color Killer Demo
Field Sync Utility Subs.Source
Field Sync Utility Subs
Field Sync Quick Test
VFFS.EMPTY.SOURCE
VFFS.EMPTY
VFFS.BOXES
VFFS.GRAPH
VFFS.GIRLS
VFFS.BYE
VFFS.LORES

Fun With Mixed Fields
LORES Colors 121
LORES1 Create
LORES2 Create

LORES1
LORES2
Gentle Scroll Set.Source
Gentle Scroll Set

Gentle Scroll Tester
Fast Background.Set.Source
Fast Background.Set
Fast Background Demo

PLUS — Two mystery "bonus"
programs!

The 26-program DOS 3.3 diskette costs only \$14.95. It is fully copyable for your personal use only and includes complete *Apple Assembler* source listings. The disk may be de-muffined or niffumed to DOS 3.2. All source listings and machine-language programs and modules are easily changed using the Apple Assembler provided on the DOS Toolkit. You can order this diskette using the attached card or directly from . . .

SYNERGETICS
Box 1300
746 First Street
Thatcher, AZ 85552
Tel: 602-428-4073

In future volumes, we will be looking at lots of exciting new ways to add variable resolution color and grey scale to your Apple, and ways to do spinners, animation, and other partial screen scrolls. We will also do in the iron statue and the golden clockwork canary once and for all. See you there.

PREVIEW OF VOLUME 2

Here's a sampler of some of the exciting new stuff you'll find in Enhancing Your Apple II, Volume 2.

DAISY DUMPERS -

Some exceptionally high quality and very fast graphics screen dumps for daisywheel printers. You get sharp and solid lines with perfectly square corners.

TEN-CENT FIX -

A quick and easy modification of your cassette recorder that will dramatically increase reliability and ease of use.

ADVENTURE EMERGENCY TOOLKIT -

Do in the iron statue and the golden clockwork canary once and for all. Learn about the three ultra-challenging "hidden" adventures buried in ALL Adventure programs.

SIX-WAY KEYBOARD IMPROVER -

Simple and super cheap add-on hardware card that will give you auto repeat, shift key mod, external keypad, lap keyboard, key duration mode, and user-defined keys. Since it goes between the existing keyboard and the encoder, it's compatible with all existing software.

SHOW-N-TELL AUTO MENU -

A very user-friendly menu system for noncomputer people. Show only what you want and where and how you want it, with full sound and animation.

FAST AND EXACT FIELD SYNC -

A low-cost hardware add-on that gives you an exact screen lock in any field. Does everything the mod of Enhancement 4 does, only it does it faster, simpler, and much more flexibly and conveniently.

VIDEO SPLITS AND WIPES -

Professional video special effects using fast exact field sync. You can smoothly wipe between screens or set up animated split-screen displays.

ULTRASONIC BSR INTERFACE -

A two-dollar hardware add-on that lets you control the world with your Apple by way of an ultrasonic link and a BSR controller.

.... PLUS MUCH MORE !

WE NEED YOUR HELP!

A response card is included in the back. Please use it to tell us the best uses that you find for your mixed fields and the best HIRES background patterns that you found, along with any requests you have for future enhancements. This card will automatically register you for future corrections and updates!

IMPORTANT — For fastest service, please send each card to the right address.

Parts kit orders go to PAIA. Everything else goes to SYNERGETICS.

Sorry, no purchase orders or billing. We also cannot ship to a foreign address.

Dealer inquiries invited.

INDEX

A

Action files, 32–33
 Adaptor socket, 23
 Add-ons and modifications, 221–226
 Address filter, 63–66
 ALTFLD subroutine, 108, 112
 Applesoft, 37, 39–41, 51, 96, 99, 121, 139, 153, 154, 178–179
 ASCII filter, 62–63
 Attack methods, 75
 Autocolor circuitry, 28
 Autostart
 monitor, 42, 46
 reset, 18, 20, 24
 ROM, 81

B

Background colors, 201–226
 Backup copies, 8
 Bar graphs, 117, 121, 153
 Base addresses, 180, 182–185, 202
 BEDGE subroutine, 107, 112
 Breakpoints, 81, 82
 “Brute force” coding, 177
 Bugs, 35
 Bulk files, 32–33, 48–49, 61, 68–69, 75

C

Cell, color, 207–208, 209, 212
 Chain, timing, 92
 Change detection, 83–84
 Changeover switch, 10, 11–12
 Character display, 172

Clock cycles, 91–93, 94, 107, 108–111
 Code(s), 84
 dead, 59
 machine-language, 33, 37
 modules, 32, 107
 pattern, 209–214
 rational, 47, 48, 58, 59–60
 source, 34
 tearing into machine-language, 29–87
 video display, 62
 working, 32
 Color(s)
 background, 201–226
 band, 24–28
 burst, 18, 19, 44, 93
 cell, 207–208, 209, 212
 combinations, 205, 206–207
 complementary, 205, 206
 decoding circuits, 118
 dots, 202–206, 207
 file, 214, 221
 HIRES, 207, 209
 killer
 automatic, 19
 programmable, 17–29
 what is?, 18
 lines, 18
 LORES, 209–210
 pastel, 212–213
 patterns, 208, 210–211
 shifted and unshifted, 203–206
 Complementary colors, 205, 206
 Control commands, software, 18
 CRUDE, 99, 107, 112, 175

D

Dead code, 59, 87
Debug programs, 31
Decoding circuits, color, 18
Delay subroutine, 110
Demo programs, 152, 153
Diagnostic helps, 81
Disassembler, 87
Display
 code, video, 62
 file, 118
Documentation, 8, 34
DOS, 38–39, 44, 51, 67, 82
 file, 38–39
 hooks, 38, 178
Dot(s)
 color, 202–206, 207
 patterns, 209–213
 positions, 204–205
Dummy soft switches, 139, 144

E

Edge detector, 107
Exact sync, 108–111, 112

F

FEDGE subroutine, 107–108, 110, 112
Field(s)
 alternator, 108, 115
 mixed, 117–144
 rate timing, 91
 switch modification, 96–98
 switching, 118
 sync, 89–115, 175
 modification, 173, 175, 196, 226
File(s)
 action and bulk, 32–33
 bulk, 48–49, 61, 68–69
 color, 214, 221
 design, 144
 display, 118
 filters, 60–61, 67, 69
 flag, 66–67
 locations, 144
 working, 121–144
Filter
 address, 63–66
 ASCII, 62–63
 file, 60–61, 67, 69
 stash, 60–61, 63
Flag file, 66–67
Force feeder, 83
Full-color mixed graphics, 18

G

Garbage, 8, 59, 90
Gentle scroll, 171–200
 flowchart, 186
 program, 178, 179, 180, 185, 187–193
Glitch(es), 160, 161, 201, 202, 226
 riddance, 150–151

Glitch(es)—cont

 stomper, 150, 151, 159–169
Glomper(s)
 of the first kind, 12–14
 of the second kind, 14–16
 two, 10, 11–16
Graph, 153
Graphics, 38, 90, 108, 112, 138, 160
 full-color mixed, 18

H

Hardware, 159–160, 172
 color killer, 19, 24–28
 modifications, 10, 89, 159, 162–163, 196
Hex dumps, 152–153, 154, 194
High
 memory pointer; *see* HIMEM
 RAM, 35, 38–39
 – Resolution Character Generator; *see* HRCG
HIMEM, 39–41, 99, 121
 pointer, 178, 179
HIRES, 9, 10, 17, 19, 24, 37, 38, 90, 112, 117, 178
 base addresses, 66
 colors, 150, 159, 207, 209
 demo program, 152
 display, 153
 graphics, 66, 196
 screen(s), 66, 75, 153
 pages, 33
 subroutines, 180, 201
 utility, 201
Hook(s), 36, 59, 81
 DOS, 178
 programs, 32
 scroll, 35, 85
Horizontal
 patterns, 136–137, 139
 – rate timing, 91, 92–93
 scan lines, 119–120, 147
 sync pulse, 92, 93
HRCG, 33–35, 44, 46, 48, 62, 75–76, 99, 172, 178–179, 202
Hues, 207, 210, 212–213

I

Integer BASIC, 37, 39–41, 51, 121
I/O sockets, 7, 27
Irrational code, 58

K

Keyboard
 buffer, 37, 38, 43, 178
 entry hooks, 36
 strobe, 41–42

L

Light pens, 89, 90, 95, 108, 111
Logic analyzer, 83

LOMEM, 39–41, 99
 Loops, 58, 176–177
 LORES, 9, 10, 17, 19, 37, 38, 90, 117, 178
 colors, 153–154, 158, 209–210
 Low
 – memory pointer; *see* LOMEM
 RAM, 35–38
 Luminance video, 93

M

Machine-language
 code, 33, 37, 138, 200
 tearing into, 29–87
 programs, 30, 32, 41, 43–44, 49, 121
 subroutine, 96, 171, 214
 Machined-pin contacts, 23
 Master timing reference, 91
 Memory
 map, 178–185
 mapping, 35
 Mixed
 fields, 117–144, 158, 159, 175
 graphics, 18, 19
 Modification(s), 19–20, 144, 150, 221–226
 hardware, 10, 26
 field-switch, 96–98
 field sync, 95–98, 173, 175
 Modulator, rf, 10, 11–12, 16, 28
 Monitor(s), 42, 46, 51, 81, 82
 system, 8
 Motion perception, 173–174
 Multiplexer, 91
 Music synthesizer card, 16

P

Partial boot, 83
 Pastel colors, 212–213
 Pattern(s)
 codes, 209–214
 color, 208, 210–211
 dots, 209–213
 Print output hooks, 36
 Program(s)
 bugs, 35
 gentle scroll, 178, 179, 180, 185,
 187–193
 hook, 32
 locations, 44–46, 51–56
 machine-language, 30, 32, 34, 41, 43–44,
 49, 121
 structure, 33
 test and debug, 31
 Programmable color killer, 17–29
 Programming, machine-language, 30–31

R

RAM
 high, 35, 38–39
 low, 35–38
 user, 46
 Raster scan collisions, 174–176

Rational code, 47, 48, 58, 59–60
 Relocatable programs, 58, 178
 Remapping, 173, 176–178, 180–181, 194
 Reset, autostart, 18, 20, 24
 Rf modulator, 10, 11–12, 16, 28

S

Scan lines, 145, 147, 151, 207, 211,
 212–213, 214
 horizontal, 119–120
 Screen switches, 41–42
 Script, 75–76
 Scroll, 35, 83, 171, 174
 gentle, 90
 hooks, 35, 85
 subroutine, 85
 Shape table, 66
 Shifted colors, 203–206, 210
 Signal, black and white, 18
 Socket(s)
 adaptor, 23
 I/O, 7, 27
 Soft switch, 118–119, 121, 138–139, 143,
 158
 Software, 8, 108, 159–160, 172
 color killer, 19, 23–24, 28
 control commands, 18
 support, 96–115, 117
 Source code, 34
 Split screens, 89
 Sprite maps, 66
 Stack, 37
 Start-of-program pointer, 41
 Stash filter, 60–61, 63
 Stashes, 32–33, 37, 59–60, 75
 Strobe, keyboard, 41–42
 Subroutine(s), 33, 49–51, 56, 64–65, 81,
 82, 99, 107–108, 111, 149, 222,
 226
 HIRES, 201
 machine-language, 96, 171, 214
 scroll, 85
 Support software, 96–115, 117
 Switch, changeover, 10, 11–12
 System monitor, 8

T

Tearing
 attack method, 34, 35
 into machine-language code, 29–87
 Test and debug programs, 31
 Text displays, 38
 Timing
 chain, 92, 94
 reference, master, 91
 video, 91
 waveforms, 91–95
 Two glompers, 10, 11–16
 TXTAB; *see* start-of-program pointer

U

Unshifted colors, 203–206, 210
Utility
 HIRES, 201
 subs, 96

V

Vertical rate timing, 93
VFFS, 121–149
Video
 display code, 62

Video—cont

Field Formatter Sub, 121; *see also* VFFS
glitch-free, 91
luminance, 93
timing, 91, 179

W

Warranty, 23
Working
 code, 32
 files, 121–144

ENHANCING YOUR APPLE II®

VOL. 1

This book is the first in an exciting new series by Don Lancaster.

- It's innovative and sneaky—digs into the innermost core of your Apple®.
- Contains fast and easy method for taking apart and understanding machine-language programs.
- Gives you hardware and software modifications that allow you to bend your Apple® to your will.
- Features programs and other hints for creating hundreds of colors or many, many patterns on your screen. Contains ideas to improve your text-on-high resolution displays. Helps you scroll your monitor screen.
- A must for the novice, the programmer, the technician, or the casual user who is still spending long hours, late nights, and many nail-biting sessions looking for IT.

HOWARD W. SAMS & CO., INC.

4300 West 62nd Street, Indianapolis, Indiana 46268 USA