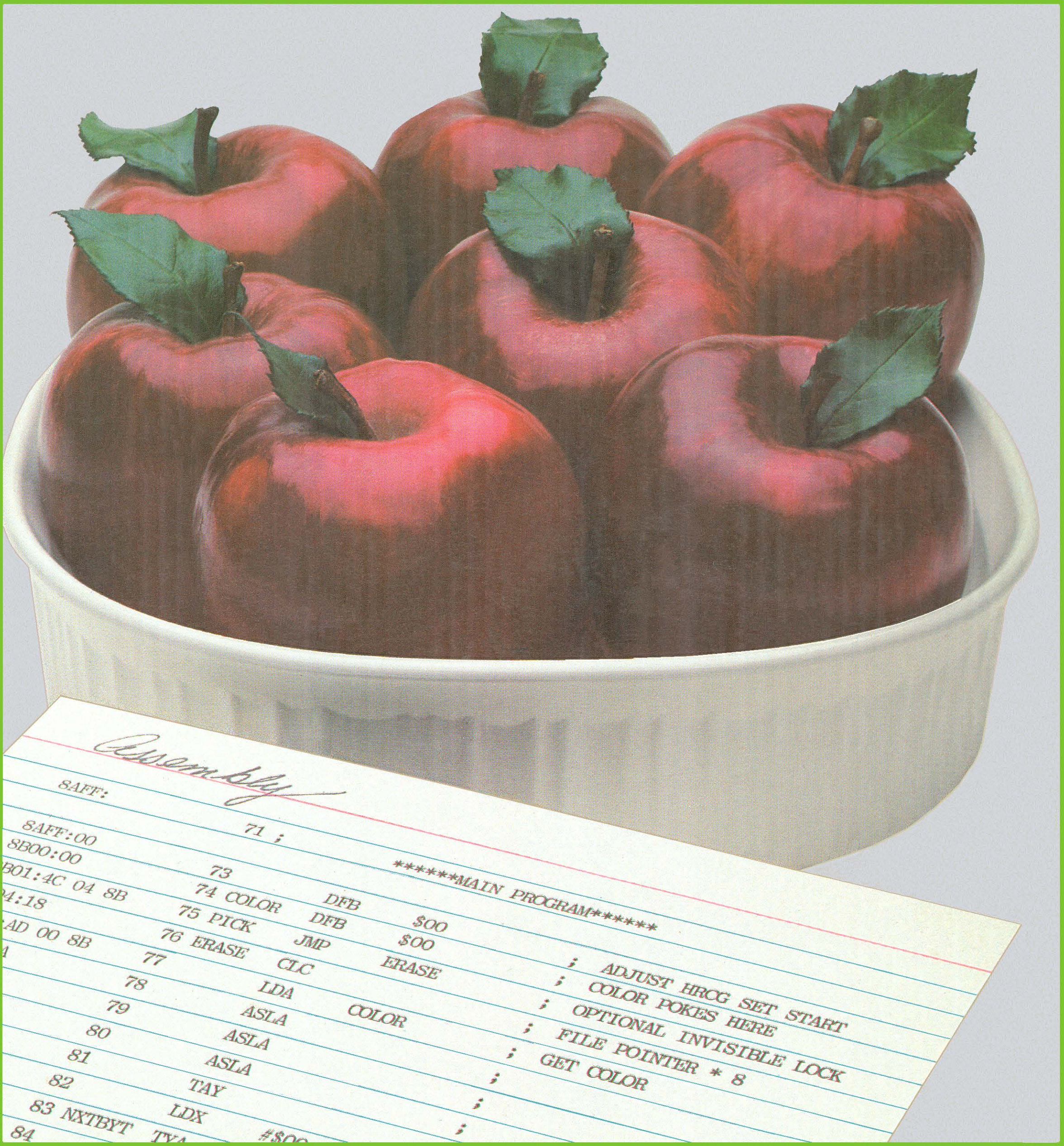
Assembly Cookbook for the Apple® II/IIe

part two

Don Lancaster



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

Assembly Cookbook for the Apple™ II/IIe

(part two)

by

Don Lancaster

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

ISBN: 978-1-882193-17-2

THIRD EDITION
FIRST PRINTING 2011

International Standard Book Number: 978-1-882193-17-2

part II
The Ripoff Modules

part two

The Ripoff Modules

by

Don Lancaster

SYNERGETICS SP PRESS

3860 West First Street, Thatcher, AZ 85552 USA

(928) 428-4073 http://www.tinaja.com

ISBN: 978-1-882193-17-2

HOW TO USE THE RIPOFF MODULES

The *Ripoff Modules* are a series of nine interactive demos designed to show you how to handle many common Apple machine language programming problems. Each module is listable, completely documented, and out in the open where you can easily access it.

I've tried to emphasize what really gets used, since just about all programming books and most program libraries center on largely outdated, cumbersome, and irrelevant dino stuff, rather than answering the real gut questions, such as "What's the best way to handle lots of text messages that might be mixed and matched together?" "Can I do a fast and well-behaved random number generator?" "Show me how to handle sound effects and musical songs;" or "How can I quickly shuffle cards or rearrange array values?"

Originally, I wanted to have lots of short demo modules. But, there are so many different important things to learn in Apple assembly language that there is simply no way to cram everything into a single book. So, instead, I decided to take the nine things that beginning assembly students seem to have the most trouble with, and expand on these in some depth.

All nine modules and bunches of other goodies are also available on a sanely priced and crammed-full companion diskette, which you can order using the card in the back of this book.

Naturally, full source code is included for each and every module. Of course, the diskette is totally unlocked, unprotected, and fully copyable. You have your choice of EDASM or S-C Assembler formats.

Most other assemblers will accept either of these formats, or else will provide a way to convert them.

A companion voice hotline service goes with this support diskette, similar to the hotline that is provided to *Enhancing* users. This support service is free, except for the usual phone charges.

You are, of course, totally free to adapt and use these ripoff modules in any way you want for any purpose. Just play fair. Give credit for any commercial use and don't try to compete head on.

Most ordinary Apple modules and subroutines are *result* oriented. This means that they are trying to get some job done as quickly and as compactly as possible. Our ripoff modules are *method* oriented instead. I have picked the modules to show you certain ways of handling different programming tasks. I've tried to make each method as mainstream and innovative as possible.

Which means that most of these modules will not have you gripping the edge of your chair in suspense, or rolling in the aisles with laughter over what they are actually doing. The modules are not intended to be arcade-quality entertainment, nor are they supposed to give you spectacular results, when used one at a time by themselves. The modules are intended instead to be a learning tool that shows you how to tackle the real gut issues involved in creating your own Apple machine language programs.

There are lots of ways you can use the ripoff modules . . .

USING THE RIPOFF MODULES

Read about them
Run them
List them
Tear them apart

Study them
Change them
Adapt them
Close the loop

Here's how to claim these modules as your own, and to add them to your own programs: First read the background text that goes with each module, so you can see what the module is intended to do. It turns out that any particular programming technique works well for some *range* of complexity, and may be overkill for simpler things and cumbersome or inefficient for very elaborate jobs. So, be sure you understand the intended use of each module, along with any simpler or more complex alternatives.

Next, run the program and watch or listen to it doing its thing. Since many of the modules will stand on their own, what they do will be pretty much limited to pointing out how they work and how they handle a certain task. In some cases, I've "trumped up" a simple example of something complex that the module is supposed to handle. Now, there may be a better way to get the *result* shown by the simple example, but, once again, that's not our point or purpose. We're after *method* here.

Then, reset to the monitor and list the program. Use the "tearing method" from Enhancement 3 of *Enhancing Your Apple II,* Volume I (Sams 21822). Color code each and every disassembled line with a page highlighter, following the *tearing* guidelines. Do this *before* you study the actual source code in depth. The reason is to gain practice reading and understanding machine language listings, particularly for those modules or programs for which you do *not* have source code.

The next step is to compare your "torn" listing against the actual source code shown here in these modules, to be sure you understand exactly what is happening when.

So much for the *analysis.* When you understand the point and purpose of each module, try some *synthesis.*

Capture a copy of the source code for the module, using an assembler of your choice. Then, make some fairly simple changes in the source code, so it will do something "alike but different somehow." Save this to a new diskette, and then assemble and run your new object code. After that, add some bells and whistles to the module's demo so it becomes longer, more interesting, or more exciting.

Now the fun begins. Rewrite the module source code one more time. Only now, make it do something you want it to do in the way you want it done, rather than the way that is shown here. Test your object code, and then actually use it in a larger program of your choosing.

Needless to say, the more time and effort you spend in understanding and capturing these modules, the more value they will be to you.

Finally, close the loop. Use the response card in back or call the hotline to let me know how you have used the existing modules and which new ones you need or would like to see.

Some of the later modules will "borrow" portions of earlier ones to keep the code simple and not reinvent the wheel. We have tried to note what is needed where. The companion diskette also includes an object code program called THE WHOLE BALL OF WAX, which combines all of the ripoff modules together all at once, along with a unifying demo.

Here's a summary of the ripoff modules and what they are intended to do . . .

RIPOFF MODULE SUMMARY

0. THE EMPTY SHELL—

A framework you can use to create most any machine language program of your choosing.

1. FILE BASED PRINTER—

The standard way to output short and fixed text messages using a common message file.

2. IMBEDDED STRING PRINTER—

A much better way to "mix and match" fixed text messages that are imbedded directly into your source code.

3. MONITOR TIME DELAY—

How to use the Apple's WAIT subroutine for animation and other system timing needs.

4. OBNOXIOUS SOUNDS—

A multiple sound effects generator that "calculates" lots of different sounds with minimum code.

5. **MUSICAL SONGS—**

   The standard "red book tones" method of making music, along with a few improvements and upgrades.

6. **OPTION PICKER—**

   How to do menu options or pick modules using the forced subroutine return method.

7. **RANDOM NUMBERS—**

   A fast and usable way to generate "random" numbers without the fatal flaws of the Applesloth "RND" code.

8. **SHUFFLE—**

   An extremely fast "random exchange" method of rearranging an array of numbers or file values.

The ripoff modules each take up one to three pages of memory. Together they sit from hex $6000 through $7300. The location of each module is shown in its source code.

If you want to interact between Applesloth and these modules, just do a HIMEM: 24575 as your first program line. This will protect the module space from being plowed. You can access the code on a PEEK and POKE basis, using your copy of *The Hexadecimal Chronicles* (Sams 21802) to show you the linking points.

One thing we have not, and will not do, is show you BASIC equivalents for the ripoff modules. The whole point of learning assembly language programming is to do so in ways that optimize the use of machine language. Thus, you *never* do something the way BASIC does. That's not even wrong. Only dumb.

On to the modules . . .

0

THE EMPTY SHELL

a framework you can use to create most any machine language program

Here are 500 lines of source code that do—absolutely nothing! It's called the *empty shell* and you use it as a framework for building your own source codes.

Actually, you'll find the empty shell doing lots of good things for you. Since it is usually much easier to edit *existing* code than to *enter* new code on most assemblers, the empty shell makes writing your custom source codes much faster. Secondly, the empty shell forces you to put decent documentation into the program ahead of time, rather than waiting until the last minute and then not doing it. This also keeps your style consistent from program to program.

The empty shell should also give you code that is far cleaner and more understandable. Finally, and most conveniently, the empty shell contains a long machine readable list of practically all of the useful Apple II and IIe subroutines and entry points. Rather than looking these up in a dozen different places, you simply *eliminate* the ones you do *not* want.

The empty shell is, of course, structure, and as we've seen, structure of any sort in a computer program is inherently despicable and evil. Nonetheless, we will use the sixteen part structure we looked at back in chapter four.

All the rest of the ripoff modules will show us examples of how the empty shell works and how to use it.

But, where do you start? . . .

To use the EMPTY SHELL.SOURCE, first eliminate what you do *not* want or need. This is best done *backward* from end to beginning.

Then, edit or change what is left to create your new source code. This is usually done *frontward* from start to finish.

First, of course, you will want to customize and personalize your own EMPTY SHELL.SOURCE by putting your own name, company, and copyright notice where mine now are. Do not rewrite to the companion diskette. Instead, save everything new on your own new diskettes. That way, you can always return to the originals if disaster strikes.

Here's some more detail on how to go about . . .

USING THE EMPTY SHELL

1. Assemble EMPTY SHELL.SOURCE and make an assembly listing hard copy. Then reload EMPTY SHELL.SOURCE into your assembler or "new way" word processor.
2. Start at the end of the source code and eliminate what you do *not* want. First, check the last line and decide whether you want to use LST OFF or LST ON. LST ON is a good choice for early program versions.
3. Decide how long your program files are to be. If you are using no DFB style files, or if you need less than 256 bytes of single-byte file values, then shorten the DFB section by deleting lines. If you need more file bytes, extend by copying.
4. Go to your hard copy and check off the hooks and constants you are going to use. If you are "old way" editing, put these in alphabetical order and then copy them to the end of their source code listings. Then delete all the unused hooks and constants.
5. Begin editing from the first line. Change the origin, then the title box. Continue editing by rewriting the "What it does," "How to use it," "Gotchas," "Enhancements," and "Random Comments." Don't worry too much about getting these perfect, since you will almost certainly change them as you edit and debug your source code.
6. Add any new hooks and constants that you want to predefine.
7. Enter your high level code and the documentation for the big lumps. Overwrite the NOPs with actual code and comments. Should you need more room, go to the assembler's insert mode and continue.
8. Do the same thing for the little lumps and the crumbs. Then enter your file values.
9. Assemble your new source code and do an assembler listing. Then repair all errors and repeat the process until you get an "error-free" result.
10. Eliminate any spurious lines and comments that may be left over from the original and reassemble.
11. Test your code, and proceed debugging from here just as you would with any other source code.

I've tried to include a fairly complete list of hooks. But note that not every hook will work on every version Apple. For instance, STEP and TRACE will only run with an old autostart ROM, while VBL and ALTCSON will only perform on an Apple IIe.

By the way, I've shortened some of the IIe labels so they are only seven or fewer characters long. You may prefer to use the "official" labels instead.

If you use any "version-specific" Apple features, be sure to include tests in your program to make sure you have the right machine in use. In general, most "mainstream" autostart programs will run on a IIe, but programs that make use of new IIe features will not work on older versions, and may even hang. If you must use some of the "oldies but goodies," it may be best to drag along the needed code *inside* your own program. Stock ID routines are included with "new" EDASM.

Should you be "new way" editing your empty shell, just delete anything unwanted or unneeded as it comes up. Once again, it is best to work from bottom to top in reverse order.

The easiest "old way" means of getting rid of extra and unwanted hooks is to copy those you need to the end of the hook listing and then delete all of the original hooks in one swell foop. With "new way" editing, just chop out what you don't need on the way by.

Since EMPTY SHELL.SOURCE is so complete, it ends up a tad long and rather slow in loading. After you have worked with it for a while, you might like to do a "short form" version of EMPTY SHELL.SOURCE that more meets your specific programming needs. If you do this, keep a printed copy of the original on hand for reference.

A tip . . .

ALWAYS do some minor fixup and pretty printing at the same time you make any important source code corrections.

Whenever you are fixing up fatal errors and making heavy changes in your source code, spend some time to clean up your documentation, improve page breaks, insert spacing, do pretty printing, eliminate unwanted lines, and cosmetic stuff like this. Each reassembly should include both heavy and light repairs.

A good goal is one line of cosmetic fix for each line of heavy fix.

This way, by the time you finally get your program debugged and working, it also will be pretty much properly documented and attractive to look at. Whatever you do, don't save the documentation for last. Start with your documentation. Sharpen, improve, clarify as you go along.

All the rest of the ripoff modules were written using the EMPTY SHELL.SOURCE. Use these as study examples, and then work up your own custom shell that meets your personal programming needs.

MIND BENDERS

— Write a WPL program that automates your empty shell setup, through use of prompts and directed questions.

— If your "new way" word processor has glossary or user-defined keys, show how to use these for single key macros and other speedup tricks.

— Solve the new way tabbing problem so that active source code lines position themselves correctly, yet comment lines remain intact.

— What tests should your source code include to make sure of . . .

—uppercase vs lowercase?
—II vs IIe?
—40 vs 80 column?
—paddles vs joystick?
—joystick orientation?

PROGRAM RM-0

THE EMPTY SHELL

```
----- NEXT OBJECT FILE NAME IS EMPTY SHELL
6000:                     ORG  $6000     ; ORIGIN GOES HERE



6000:               ;   *****************************************
6000:               ;   *                                       *
6000:               ;   *         -< THE  EMPTY  SHELL >-       *
6000:               ;   *                                       *
6000:               ;   *            (DUMMY PROGRAM)            *
6000:               ;   *                                       *
6000:               ;   *      VERSION 1.0  ($6000-$6160)       *
6000:               ;   *                                       *
6000:               ;   *               5-24-83                 *
6000:               ;   *.......................................*
6000:               ;   *                                       *
6000:               ;   *          COPYRIGHT C 1983 BY          *
6000:               ;   *                                       *
6000:               ;   *     DON LANCASTER AND SYNERGETICS     *
6000:               ;   *     BOX 1300, THATCHER AZ., 85552     *
6000:               ;   *                                       *
6000:               ;   *     ALL COMMERCIAL RIGHTS RESERVED    *
6000:               ;   *                                       *
6000:               ;   *****************************************


6000:               ;          *** WHAT IT DOES ***

6000:               ;   THIS PROGRAM IS A DUMMY SHELL USED AS A STARTING
6000:               ;   POINT FOR YOUR OWN ASSEMBLY LANGUAGE PROGRAMS.
6000:               ;
6000:               ;
6000:               ;
6000:               ;


6000:               ;          *** HOW TO USE IT ***

6000:               ;   TO USE, EDIT THE PROGRAM BY MOVING THE ORIGIN,
6000:               ;   CHANGING THE TITLE, REMOVING EXTRA EQU'S, ADDING
6000:               ;   YOUR OWN WORKING CODE, ALTERING THE DATA FILES
6000:               ;   AND DOING WHATEVER ELSE MAY BE NEEDED TO BUILD
6000:               ;   YOUR OWN CUSTOM ASSEMBLED PROGRAM OR MODULE.
6000:               ;
```

PROGRAM RM-0, CONT'D . . .

```
6000:          ;             *** GOTCHAS ***

6000:          ;   ANYTHING ESSENTIAL FOR USE OR UNDERSTANDING OF THE
6000:          ;   PROGRAM GETS PUT HERE.  THIS INCLUDES SPECIAL NEEDS
6000:          ;   SUCH AS EXTRA MEMORY, ANY COMPANION CODE MODULES, OR
6000:          ;   ANY SPECIAL HARDWARE.
6000:          ;
6000:          ;


6000:          ;            *** ENHANCEMENTS ***

6000:          ;   PUT ANY ADD-ONS, "EXTRA TRICKS", OR SPECIAL
6000:          ;   USES HERE.  INCLUDE USE TIPS AND APPLICATIONS.
6000:          ;
6000:          ;
6000:          ;
6000:          ;


6000:          ;            *** RANDOM COMMENTS ***

6000:          ;   IF THERE IS SOMETHING ELSE YOU WANT TO SAY THAT'S
6000:          ;   NOT ALL THAT IMPORTANT, YOU CAN ADD IT IN THIS SPACE.
6000:          ;
6000:          ;
6000:          ;
6000:          ;
```

PROGRAM RM-0, CONT'D . . .

```
6000:      ;           *** HOOKS ***

0020:      WNDLFT   EQU  $20       ; SCROLL WINDOW LEFT
0021:      WNDWDTH  EQU  $21       ; SCROLL WINDOW WIDTH
0022:      WNDTOP   EQU  $22       ; SCROLL WINDOW TOP
0023:      WNDBOT   EQU  $23       ; SCROLL WINDOW BOTTOM
0024:      CH       EQU  $24       ; CURSOR HORIZONTAL
0025:      CV       EQU  $25       ; CURSOR VERTICAL
0026:      GBASL    EQU  $26       ; LORES BASE LOW
0027:      GBASH    EQU  $27       ; LORES BASE HIGH
0028:      BASL     EQU  $28       ; TEXT BASE LOW
0029:      BASH     EQU  $29       ; TEXT BASE HIGH
002C:      HEND     EQU  $2C       ; LORES RIGHT END H LINE
002D:      VBOT     EQU  $2D       ; LORES BOTTOM OF V LINE
0030:      COLOR    EQU  $30       ; LORES COLOR
0031:      INVFLG   EQU  $31       ; NORMAL/INVERSE/FLASH (FF,7F,3F)
0033:      PROMPT   EQU  $33       ; HOLDS PROMPT SYMBOL
0036:      CSWL     EQU  $36       ; OUTPUT CHARACTER HOOK LOW
0037:      CSWH     EQU  $37       ; OUTPUT CHARACTER HOOK HIGH
0038:      KSWL     EQU  $38       ; INPUT CHARACTER HOOOK LOW
0039:      KSWH     EQU  $39       ; INPUT CHARACTER HOOK HIGH
004E:      RNDL     EQU  $4E       ; RANDOM NUMBER LOW
004F:      RNDH     EQU  $4F       ; RANDOM NUMBER HIGH

0100:      STACK    EQU  $0100     ; STACK PAGE ACCESS

0200:      KEYBUF   EQU  $0200     ; KEYBUFFER START

03D0:      DOSWRM   EQU  $03D0     ; DOS WARM START JMP
03D3:      DOSCLD   EQU  $03D3     ; DOS COLD START JMP
03D6:      DOSFLM   EQU  $03D6     ; DOS FILE MANAGER JUMP
03D9:      DOSRWTS  EQU  $03D9     ; DOS RWTS JUMP
03DC:      DOSIPRM  EQU  $03DC     ; DOS FILE PARAMETER FIND JUMP
03E3:      DOSRWLS  EQU  $03E3     ; DOS RWTS PARAMETER FIND JUMP
03EA:      DOSHOOK  EQU  $03EA     ; DOS HOOK RECONNECT JUMP
03F0:      BRK      EQU  $03F0     ; BREAK ADDRESS (AUTOSTART& 2E ONLY!)
03F2:      SOFTEV   EQU  $03F2     ; SOFT RESET (AUTOSTART & 2E ONLY!)
03F4:      PWRDUP   EQU  $03F4     ; WARM START EOR CHECKSUM
03F5:      AMPERV   EQU  $03F5     ; APPLESOFT "&" JUMP
03F8:      USRADR   EQU  $03F8     ; CTRL-Y JUMP
03FB:      NMI      EQU  $03FB     ; NON-MASKABLE INTERRUPT JUMP
03FE:      IRQLOC   EQU  $03FE     ; INTERRUPT ADDRESS LOW

0400:      TEXTP1   EQU  $0400     ; START OF TEXT PAGE ONE
0800:      TEXTP2   EQU  $0800     ; START OF TEXT PAGE TWO
2000:      HIRESP1  EQU  $2000     ; START OF HIRES PAGE ONE
4000:      HIRESP2  EQU  $4000     ; START OF HIRES PAGE TWO
```

PROGRAM RM-0, CONT'D . . .

```
C000:          IOADR   EQU  $C000     ; KEYBOARD INPUT
C010:          KBDSTR  EQU  $C010     ; KEYSTROBE RESET
C020:          TAPEOUT EQU  $C020     ; CASSETTE OR AUDIO OUT
C030:          SPKR    EQU  $C030     ; SPEAKER CLICK OUTPUT
C040:          STROBE  EQU  $C040     ; GAME CONNECTOR STROBE
C050:          TXTCLR  EQU  $C050     ; GRAPHICS ON SOFT SWITCH
C051:          TXTSET  EQU  $C051     ; TEXT ON SOFT SWITCH
C052:          MIXCLR  EQU  $C052     ; FULL SCREEN SOFT SWITCH
C053:          MIXSET  EQU  $C053     ; MIXED SCREEN SOFT SWITCH
C054:          LOWSCR  EQU  $C054     ; PAGE ONE SOFT SWITCH
C055:          HISCR   EQU  $C055     ; PAGE TWO SOFT SWITCH
C056:          LORES   EQU  $C056     ; LORES SOFT SWITCH
C057:          HIRES   EQU  $C057     ; HIRES SOFT SWITCH
C060:          PB4     EQU  $C060     ; CASS IN + "FOURTH" PB INPUT "SW3"
C061:          PB1     EQU  $C061     ; OPEN APPLE + "FIRST" PB INPUT "SW0"
C062:          PB2     EQU  $C062     ; CLOSED APPLE + "SECOND" PB INPUT "S
C063:          PB3     EQU  $C063     ; "THIRD"  PUSHBUTTON INPUT "SW2"
C064:          PDL0    EQU  $C064     ; GAME PADDLE 0 ANALOG IN
C065:          PDL1    EQU  $C065     ; GAME PADDLE 1 ANALOG IN
C066:          PDL2    EQU  $C066     ; GAME PADDLE 2 ANALOG IN
C067:          PDL3    EQU  $C067     ; GAME PADDLE 3 ANALOG IN
C070:          PTRIG   EQU  $C070     ; ANALOG PADDLE RESET

C080:          STEP00  EQU  $C080     ; DISK STEPPER PHASE 0 OFF
C081:          STEP01  EQU  $C081     ; DISK STEPPER PHASE 0 ON
C082:          STEP10  EQU  $C082     ; DISK STEPPER PHASE 1 OFF
C083:          STEP11  EQU  $C083     ; DISK STEPPER PHASE 1 ON
C084:          STEP20  EQU  $C084     ; DISK STEPPER PHASE 2 OFF
000C:          STEP21  EQU  $C085     ; DISK STEPPER PHASE 2 ON
C086:          STEP30  EQU  $C086     ; DISK STEPPER PHASE 3 OFF
C087:          STEP31  EQU  $C087     ; DISK STEPPER PHASE 3 ON
C088:          MOTON   EQU  $C088     ; DISK MAIN MOTOR OFF
C089:          MOTOFF  EQU  $C089     ; DISK MAIN MOTOR ON
C08A:          DRV0EN  EQU  $C08A     ; DISK ENABLE DRIVE 1
C08B:          DRV1EN  EQU  $C08B     ; DISK ENABLE DRIVE 2
C08C:          Q6CLR   EQU  $C08C     ; DISK Q6 CLEAR
C08D:          Q6SET   EQU  $C08D     ; DISK Q6 SET
C08E:          Q7CLR   EQU  $C08E     ; DISK Q7 CLEAR
C08F:          Q7SET   EQU  $C08F     ; DISK Q7 SET

E000:          BASICLD EQU  $E000     ; ENTER BASIC COLD
E003:          BASICWM EQU  $E003     ; RE-ENTER BASIC WARM

F3D8:          HGR2    EQU  $F3D8     ; APPLESOFT CLEAR TO HIRES 2
F3E2:          HGR     EQU  $F3E2     ; APPLESOFT CLEAR TO HIRES 1
F3F4:          BKGND   EQU  $F3F4     ; APPLESOFT HIRES BACKGROUND CLEAR
F6F0:          HCOLOR  EQU  $F6F0     ; APPLESOFT HIRES COLOR SELECT
F411:          HPOSN   EQU  $F411     ; APPLESOFT HIRES POSITION
F457:          HPLOT   EQU  $F457     ; APPLESOFT HIRES PLOT
```

PROGRAM RM-0, CONT'D . . .

```
F800:       PLOT     EQU  $F800     ; PLOT LORES BLOCK
F819:       HLINE    EQU  $F819     ; HORIZ LORES LINE
F828:       VLINE    EQU  $F828     ; VERTICAL LORES LINE
F832:       CLRSCR   EQU  $F832     ; CLEAR FULL LORES SCREEN
F836:       CLRTOP   EQU  $F836     ; CLEAR TOP LORES SCREEN
F847:       GBSCALC  EQU  $F847     ; LORES BASE CALCULATION
F85F:       NEXTCOL  EQU  $F85F     ; INCREASE LORES COLOR BY 3
F864:       SETCOL   EQU  $F864     ; SET LORES COLOR
F871:       SCRN     EQU  $F871     ; READ LORES SCREEN COLOR
F941:       PRNTAX   EQU  $F941     ; OUTPUT A THEN X AS HEX
F948:       PRBLNK   EQU  $F948     ; OUTPUT 3 SPACES VIA HOOKS
F94A:       PRBL2    EQU  $F94A     ; OUTPUT X BLANKS VIA HOOKS

FAD7:       REGDSP   EQU  $FAD7     ; DISPLAY WORKING REGISTERS
FB1E:       PREAD    EQU  $FB1E     ; READ GAME PADDLE X
FB2F:       INIT     EQU  $FB2F     ; INITIALIZE TEXT SCREEN
FB93:       SETTXT   EQU  $FB93     ; SET UP TEXT SCREEN (NOT 2E!)
FB40:       SETGR    EQU  $FB40     ; SET UP GRAPHICS SCREEN
FB4B:       SETWND   EQU  $FB4B     ; SET NORMAL TEXT WINDOW
FBC1:       BASCALC  EQU  $FBC1     ; CALCULATE TEXT BASE ADDRESS (NOT 2E!)
FBD9:       BELL1    EQU  $FBD9     ; BEEP SPEAKER IF CTRL-G
FBE4:       BELL2    EQU  $FBE4     ; BEEP SPEAKER ONCE
FBF4:       ADVANCE  EQU  $FBF4     ; TEXT CURSOR ONE TO RIGHT
FBFD:       VIDOUT   EQU  $FBFD     ; OUTPUT ASCII TO SCREEN ONLY

FC10:       BS       EQU  $FC10     ; BACKSPACE SCREEN
FC1A:       UP       EQU  $FC1A     ; MOVE SCREEN CURSOR UP ONE LINE
FC22:       VTAB     EQU  $FC22     ; VERTICAL SCREEN TAB USING CV
FC24:       VTABA    EQU  $FC24     ; VERTICAL SCREEN TAB USING A
FC66:       ESC1     EQU  $FC66     ; PROCESS ESCAPE CURSOR MOVES
FC42:       CLREOP   EQU  $FC42     ; CLEAR TO END OF PAGE
FC58:       HOME     EQU  $FC58     ; CLEAR TEXT SCREEN AND HOME CURSOR
FC62:       CR       EQU  $FC62     ; CARRIAGE RETURN TO SCREEN
FC66:       LF       EQU  $FC66     ; LINEFEED TO SCREEN ONLY
FC70:       SCROLL   EQU  $FC70     ; SCROLL TEXT SCREEN UP ONE
FC9C:       CLEOL    EQU  $FC9C     ; CLEAR TEXT TO END OF LINE
FCA8:       WAIT     EQU  $FCA8     ; TIME DELAY SET BY ACCUMULATOR
FD0C:       RDKEY    EQU  $FD0C     ; GET INPUT CHARACTER VIA HOOKS
FD1B:       KEYIN    EQU  $FD1B     ; READ THE APPLE KEYBOARD
FD35:       RDCHAR   EQU  $FD35     ; GET KEY AND PROCESS ESC A-F
FD62:       CANCEL   EQU  $FD62     ; CANCEL KEYBOARD LINE ENTRY
FD67:       GETLNZ   EQU  $FD67     ; CR THEN GET KEYBOARD INPUT LINE
FD6A:       GETLN    EQU  $FD6A     ; GET KEYBOARD INPUT LINE
FD6F:       GETLN1   EQU  $FD6F     ; GET KBD INPUT, NO PROMPT
FD8B:       CROUT1   EQU  $FD8B     ; CLEAR EOL THEN CR VIA HOOKS
FD8E:       CROUT    EQU  $FD8E     ; OUTPUT CR VIA HOOKS
FDDA:       PRBYTE   EQU  $FDDA     ; OUTPUT FULL A IN HEX TO HOOKS
FDE3:       PRHEX    EQU  $FDE3     ; OUTPUT LOW A IN HEX TO HOOKS
FDED:       COUT     EQU  $FDED     ; OUTPUT CHARACTER VIA HOOKS
FDF0:       COUT1    EQU  $FDF0     ; OUTPUT CHARACTER TO SCREEN
```

PROGRAM RM-0, CONT'D . . .

```
FE2C:          MOVE    EQU  $FE2C     ; MOVE BLOCK OF MEMORY
FE36:          VERIFY  EQU  $FE36     ; VERIFY BLOCK OF MEMORY
FE5E:          LIST    EQU  $FE5E     ; DISASSEMBLE 20 INSTRUCTIONS
FE63:          LIST2   EQU  $FE63     ; DISASSEMBLE "A" INSTRUCTIONS
FE80:          SETINV  EQU  $FE80     ; PRINT INVERSE TEXT TO SCREEN
FE84:          SETNORM EQU  $FE84     ; PRINT NORMAL TEXT TO SCREEN
FE93:          SETVID  EQU  $FE93     ; GRAB OUTPUT HOOKS FOR SCREEN
FEB0:          XBASIC  EQU  $FEB0     ; GO BASIC, DESTROYING OLD
FEB3:          BASCON  EQU  $FEB3     ; GO BASIC, CONTINUING OLD
FEC2:          TRACE   EQU  $FEC2     ; START TRACING (OLD ROM ONLY!)
FEC4:          STEP    EQU  $FEC4     ; SINGLE STEP (OLD ROM ONLY!)
FECD:          WRITE   EQU  $FECD     ; WRITE TO CASSETTE TAPE
FEFD:          READ    EQU  $FEFD     ; READ FROM CASSETTE TAPE
FF2D:          PRERR   EQU  $FF2D     ; PRINT "ERR" TO OUTPUT HOOK
FF3A:          BELL    EQU  $FF3A     ; OUTPUT BELL TO HOOKS
FF3F:          IORESR  EQU  $FF3F     ; RESTORE ALL WORKING REGISTERS
FF4A:          IOSAVE  EQU  $FF4A     ; SAVE ALL WORKING REGISTERS
FF58:          RETURN  EQU  $FF58     ; "GUARANTEED" RETURN
FF59:          OLDRST  EQU  $FF59     ; OLD RESET, NO AUTOSTART
FF65:          MON     EQU  $FF65     ; ENTER MONITOR AND BEEP SPEAKER
FF69:          MONZ    EQU  $FF69     ; ENTER MONITOR QUIETLY
FFA7:          GETNUM  EQU  $FFA7     ; ASCII TO HEX IN 3E & 3F



6000:          ;              *** HOOKS FOR 2E ONLY! ***


C000:          CLR80CO EQU  $C000     ; 80 STORE OFF (WRITE ONLY)
C001:          SET80CO EQU  $C001     ; 80 STORE ON  (WRITE ONLY)
C002:          RAMRDMN EQU  $C002     ; READ MAIN MEMORY (WRITE ONLY)
C003:          RAMRDAX EQU  $C003     ; READ AUXILIARY MEMORY (WRITE ONLY)
C004:          RAMWRMN EQU  $C004     ; WRITE MAIN MEMORY (WRITE ONLY)
C005:          RAMWRAX EQU  $C005     ; WRITE AUXILIARY MEMORY (WRITE ONLY)
C006:          SLOTXRM EQU  $C006     ; INTERNAL ROM AT CX00 (WRITE ONLY)
C007:          SLOTXEX EQU  $C007     ; SLOT ROM AT CX00 (WRITE ONLY)


C008:          MAINZP  EQU  $C008     ; USE MAIN ZERO PAGE (WRITE ONLY)
C009:          ALTZP   EQU  $C009     ; USE ALTERNATE ZERO PAGE (WRITE ONLY)
C00A:          SLOT3RM EQU  $C00A     ; SLOT #3 INTERNAL ROM (WRITE ONLY)
C00B:          SLOT3EX EQU  $C00B     ; SLOT #3 EXTERNAL ROM (WRITE ONLY)
C00C:          OFF80CL EQU  $C00C     ; TURN 80 COLUMN OFF (WRITE ONLY)
C00D:          ON80COL EQU  $C00D     ; TURN 80 COLUMN ON (WRITE ONLY)
C00E:          ALTCSOF EQU  $C00E     ; USE MAIN CHARACTER SET (WRITE ONLY)
C00F:          ALTCSON EQU  $C00F     ; USE ALT CHARACTER SET (WRITE ONLY)
```

PROGRAM RM-0, CONT'D . . .

```
C013:        RAMRDS  EQU  $C013    ; READ RAMREAD SWITCH (READ ONLY)
C014:        RAMWTS  EQU  $C014    ; READ RAMWRITE SWITCH (READ ONLY)
C015:        SLTCXS  EQU  $C015    ; READ SLOT CX SWITCH (READ ONLY)
C016:        ALTZPS  EQU  $C016    ; READ ZERO PAGE SWITCH (READ ONLY)
C017:        SLTC3S  EQU  $C017    ; READ SLOT C3 SWITCH (READ ONLY)

C018:        S80STR  EQU  $C018    ; READ 80STORE SWITCH (READ ONLY)
C019:        VBL     EQU  $C019    ; VERT. BLANKING >80=BLANK (READ ONLY)
C01A:        TEXTS   EQU  $C01A    ; READ TEXT SWITCH (READ ONLY)
C01B:        MIXEDS  EQU  $C01B    ; READ MIXED GR SWITCH (READ ONLY)
C01C:        PAGE2S  EQU  $C01C    ; READ PAGE 2 SWITCH (READ ONLY)
C01D:        HIRESS  EQU  $C01D    ; READ HIRES SWITCH (READ ONLY)
C01E:        ALTCSS  EQU  $C01E    ; READ ALTCHAR SET SWITCH (READ ONLY)
C01F:        S80COL  EQU  $C01F    ; READ 80 COLUMN SWITCH (READ ONLY)


C080:        RB2RAM  EQU  $C080    ; READ BANK 2 RAM
C081:        WB2RAM  EQU  $C081    ; WRITE BANK 2 RAM, READ ROM
C082:        RROM    EQU  $C082    ; READ ROM ONLY, NO WRITE
C083:        RWRAM2  EQU  $C083    ; READ & WRITE RAM2 (HIT TWICE!)
C088:        RRAM1   EQU  $C088    ; READ BANK1 RAM
C089:        WRAM1   EQU  $C089    ; WRITE BANK1 RAM, READ ROM
C08A:        RB1ROM  EQU  $C08A    ; READ BANK1 ROM
C08B:        RWRAM1  EQU  $C08B    ; READ & WRITE RAM1 (HIT TWICE!)



6000:        ;                *** CONSTANTS ***
6000:        ;           *** TEXTFILE COMMANDS ***

0088:        B       EQU  $88      ; BACKSPACE
008D:        C       EQU  $8D      ; CARRIAGE RETURN
0084:        D       EQU  $84      ; DOS ATTENTION
008C:        F       EQU  $8C      ; FORMFEED
0087:        G       EQU  $87      ; RING GONG
008A:        L       EQU  $8A      ; LINEFEED
0060:        P       EQU  $60      ; FLASHING PROMPT
0000:        X       EQU  $00      ; END OF MESSAGE
```

PROGRAM RM-0, CONT'D . . .

```
6000:          ;          *** BIG LUMPS ***
6000:          ;        *** MAIN PROGRAM ***
6000:          ;       *** HIGH LEVEL CODE ***


6000:          ;       ADD ANY COMMENTS HERE THAT ARE
6000:          ;       SPECIFIC TO THE BIG LUMPS.
6000:          ;
6000:          ;
6000:          ;
6000:          ;

6000:EA        START1  NOP                ; YOUR HIGH LEVEL CODE STARTS HERE
6001:EA                NOP                ;  AND GOES ON AS FAR AS NEEDED.
6002:EA                NOP                ;
6003:EA                NOP                ;
6004:EA                NOP                ;
6005:EA                NOP                ;
6006:EA                NOP                ;
6007:EA                NOP                ;

6008:EA                NOP                ;
6009:EA                NOP                ;
600A:EA                NOP                ;
600B:EA                NOP                ;
600C:EA                NOP                ;
600D:EA                NOP                ;
600E:EA                NOP                ;
600F:EA                NOP                ;

6010:EA                NOP                ;
6011:EA                NOP                ;
6012:EA                NOP                ;
6013:EA                NOP                ;
6014:EA                NOP                ;
6015:EA                NOP                ;
6016:EA                NOP                ;
6017:EA                NOP                ;

6018:EA                NOP                ;
6019:EA                NOP                ;
601A:EA                NOP                ;
601B:EA                NOP                ;
601C:EA                NOP                ;
601D:EA                NOP                ;
601E:EA                NOP                ;
601F:EA                NOP                ;
```

PROGRAM RM-0, CONT'D . . .

```
6020:          ;            *** LITTLE  LUMPS ***
6020:          ;          *** HEAVY SUBROUTINE ***
6020:          ;        *** SUPPORTING  MODULE ***

6020:          ;          ADD ANY COMMENTS HERE THAT ARE
6020:          ;          SPECIFIC TO THE LITTLE LUMPS.
6020:          ;
6020:          ;
6020:          ;
6020:          ;

6020:EA        START2  NOP            ; YOUR MEDIUM LEVEL CODE STARTS
6021:EA                NOP            ;  HERE AND GOES ON AS FAR AS
6022:EA                NOP            ;  NEEDED.
6023:EA                NOP            ;
6024:EA                NOP            ;
6025:EA                NOP            ;
6026:EA                NOP            ;
6027:EA                NOP            ;

6028:EA                NOP            ;
6029:EA                NOP            ;
602A:EA                NOP            ;
602B:EA                NOP            ;
602C:EA                NOP            ;
602D:EA                NOP            ;
602E:EA                NOP            ;
602F:EA                NOP            ;

6030:EA                NOP            ;
6031:EA                NOP            ;
6032:EA                NOP            ;
6033:EA                NOP            ;
6034:EA                NOP            ;
6035:EA                NOP            ;
6036:EA                NOP            ;
6037:EA                NOP            ;

6038:EA                NOP            ;
6039:EA                NOP            ;
603A:EA                NOP            ;
603B:EA                NOP            ;
603C:EA                NOP            ;
603D:EA                NOP            ;
603E:EA                NOP            ;
603F:EA                NOP            ;
```

PROGRAM RM-0, CONT'D . . .

```
6040:         ;             ***  STASH   ***
6040:         ;            *** THE CRUMBS ***
6040:         ;           *** DETAIL  SUBS ***


6040:         ;         ADD ANY COMMENTS HERE THAT
6040:         ;         ARE SPECIFIC TO THE CRUMBS.
6040:         ;
6040:         ;
6040:         ;
6040:         ;

6040:EA       START3  NOP                ; YOUR LOW LEVEL CODE STARTS HERE AND
6041:EA               NOP                ;  INCLUDES ANY SHORT FILES THAT ARE
6042:EA               NOP                ;  RARELY CHANGED.
6043:EA               NOP                ;
6044:EA               NOP                ;
6045:EA               NOP                ;
6046:EA               NOP                ;
6047:EA               NOP                ;

6048:EA               NOP                ;
6049:EA               NOP                ;
604A:EA               NOP                ;
604B:EA               NOP                ;
604C:EA               NOP                ;
604D:EA               NOP                ;
604E:EA               NOP                ;
604F:EA               NOP                ;

6050:EA               NOP                ;
6051:EA               NOP                ;
6052:EA               NOP                ;
6053:EA               NOP                ;
6054:EA               NOP                ;
6055:EA               NOP                ;
6056:EA               NOP                ;
6057:EA               NOP                ;

6058:EA               NOP                ;
6059:EA               NOP                ;
605A:EA               NOP                ;
605B:EA               NOP                ;
605C:EA               NOP                ;
605D:EA               NOP                ;
605E:EA               NOP                ;
605F:EA               NOP                ;
```

PROGRAM RM-0, CONT'D . . .

```
6060:              ;              *** MAIN FILES ***

6060:              ;        ADD ANY COMMENTS HERE THAT ARE
6060:              ;        SPECIFIC TO THE MAIN FILES.
6060:              ;
6060:              ;
6060:              ;
6060:              ;

6060:00 00 00      FILE1    DFB             $00,$00,$00,$00,$00,$00,$00,$00
6063:00 00 00
6066:00 00
6068:00 00 00      FILE2    DFB             $00,$00,$00,$00,$00,$00,$00,$00
606B:00 00 00
606E:00 00
6070:00 00 00      FILE3    DFB             $00,$00,$00,$00,$00,$00,$00,$00
6073:00 00 00
6076:00 00
6078:00 00 00      FILE4    DFB             $00,$00,$00,$00,$00,$00,$00,$00
607B:00 00 00
607E:00 00
6080:00 00 00      FILE5    DFB             $00,$00,$00,$00,$00,$00,$00,$00
6083:00 00 00
6086:00 00
6088:00 00 00      FILE6    DFB             $00,$00,$00,$00,$00,$00,$00,$00
608B:00 00 00
608E:00 00
6090:00 00 00      FILE7    DFB             $00,$00,$00,$00,$00,$00,$00,$00
6093:00 00 00
6096:00 00
6098:00 00 00      FILE8    DFB             $00,$00,$00,$00,$00,$00,$00,$00
609B:00 00 00
609E:00 00
60A0:00 00 00      FILE9    DFB             $00,$00,$00,$00,$00,$00,$00,$00
60A3:00 00 00
60A6:00 00
60A8:00 00 00      FILE10   DFB             $00,$00,$00,$00,$00,$00,$00,$00
60AB:00 00 00
60AE:00 00
60B0:00 00 00      FILE11   DFB             $00,$00,$00,$00,$00,$00,$00,$00
60B3:00 00 00
60B6:00 00
60B8:00 00 00      FILE12   DFB             $00,$00,$00,$00,$00,$00,$00,$00
60BB:00 00 00
60BE:00 00
60C0:00 00 00      FILE13   DFB             $00,$00,$00,$00,$00,$00,$00,$00
60C3:00 00 00
60C6:00 00
60C8:00 00 00      FILE14   DFB             $00,$00,$00,$00,$00,$00,$00,$00
60CB:00 00 00
60CE:00 00
```

PROGRAM RM-0, CONT'D . . .

```
60D0:00 00 00  474 FILE15  DFB            $00,$00,$00,$00,$00,$00,$00,$00
60D3:00 00 00
60D6:00 00
60D8:00 00 00  475 FILE16  DFB            $00,$00,$00,$00,$00,$00,$00,$00
60DB:00 00 00
60DE:00 00
60E0:00 00 00  476 FILE17  DFB            $00,$00,$00,$00,$00,$00,$00,$00
60E3:00 00 00
60E6:00 00
60E8:00 00 00  477 FILE18  DFB            $00,$00,$00,$00,$00,$00,$00,$00
60EB:00 00 00
60EE:00 00
60F0:00 00 00  478 FILE19  DFB            $00,$00,$00,$00,$00,$00,$00,$00
60F3:00 00 00
60F6:00 00
60F8:00 00 00  479 FILE20  DFB            $00,$00,$00,$00,$00,$00,$00,$00
60FB:00 00 00
60FE:00 00
6100:00 00 00  480 FILE21  DFB            $00,$00,$00,$00,$00,$00,$00,$00
6103:00 00 00
6106:00 00
6108:00 00 00  481 FILE22  DFB            $00,$00,$00,$00,$00,$00,$00,$00
610B:00 00 00
610E:00 00
6110:00 00 00  482 FILE23  DFB            $00,$00,$00,$00,$00,$00,$00,$00
6113:00 00 00
6116:00 00
6118:00 00 00  483 FILE24  DFB            $00,$00,$00,$00,$00,$00,$00,$00
611B:00 00 00
611E:00 00
6120:00 00 00  484 FILE25  DFB            $00,$00,$00,$00,$00,$00,$00,$00
6123:00 00 00
6126:00 00
6128:00 00 00  485 FILE26  DFB            $00,$00,$00,$00,$00,$00,$00,$00
612B:00 00 00
612E:00 00
6130:00 00 00  486 FILE27  DFB            $00,$00,$00,$00,$00,$00,$00,$00
6133:00 00 00
6136:00 00
6138:00 00 00  487 FILE28  DFB            $00,$00,$00,$00,$00,$00,$00,$00
613B:00 00 00
613E:00 00
6140:00 00 00  488 FILE29  DFB            $00,$00,$00,$00,$00,$00,$00,$00
6143:00 00 00
6146:00 00
6148:00 00 00  489 FILE30  DFB            $00,$00,$00,$00,$00,$00,$00,$00
614B:00 00 00
614E:00 00
6150:00 00 00  490 FILE31  DFB            $00,$00,$00,$00,$00,$00,$00,$00
6153:00 00 00
6156:00 00
6158:00 00 00  491 FILE32  DFB            $00,$00,$00,$00,$00,$00,$00,$00
615B:00 00 00
615E:00 00
```

PROGRAM RM-0, CONT'D . . .

```
6160:          494 ;          *** BOTTOM LINE COMMENTS ***

6160:          496 ;        ADD ANY FINAL COMMENTS YOU FEEL
6160:          497 ;        ARE NEEDED IN THIS SPACE.



*** SUCCESSFUL ASSEMBLY: NO ERRORS
```

1

FILE BASED PRINTER

the "standard" way to output short and fixed text messages by using a common message file.

Outputting text is probably the most fundamental and most important task we would ever ask of a machine language Apple program. You might want to use the text to create a printed record, to inform the user via the video screen, or to pass a command to the disk system.

It turns out that there is no "best" way to go about outputting text from machine language. Instead, there are many different methods you can pick. These methods are based on *how many* messages you must output, on *how long* each message is, and on *how changeable* the messages have to be.

Further, you have to decide just *where* your message is going to go as well. Usually, to output a character, you get it from somewhere and put it in the accumulator. Then you go to a text outputting subroutine that puts the character where you want it to appear. You continue this until some change occurs, such as a marker or length count. Then you go on to the next task at hand.

Here are some possible . . .

PLACES TO OUTPUT TEXT

Direct store to the text screen
To COUT hook subroutine $FDF0
To COUT1 screen subroutine $FDED
To a HIRES character generator
To your own custom code

If you *direct store* to a screen location, you end up putting characters on the screen in the shortest possible time, and you are always sure *exactly* where on the screen the character is to go. As an example, a $C1 stored in $0400 puts an uppercase "A" in the upper left-hand screen position. But, the screen locations aren't mapped in an obvious order, and you get into real hassles over carriage returns and scrolls. There is also no simple way to get a hard copy of a direct screen store. So, direct storing to the screen is usually limited to game scores, status lines, and special effects, rather than being a mainstream way of doing things.

Since outputting text is so important, there are two subroutines built into the Apple's monitor, designed to do most text outputting tasks in the way that most people want them done. One subroutine is called COUT and is located at $FDED. This subroutine will output characters to anything that is connected to the Apple by way of two *character hooks* called CSWL and CSWH and located at $0036 and $0037.

Normally, DOS grabs these character hooks so that it can intercept all output commands, just in case there is something intended for the disk. In turn, DOS will take whatever was plugged into the output hooks, and then plug these into itself.

For instance, a normal hard copy character will get routed from your code to COUT, where it gets passed on to DOS, which checks it for disk commands. The character is then passed on to a printer card, whose code often begins at location $C100. The code will then send the proper commands to the printer itself to print the character. Finally, if you want it to, the printer card code will echo the character on to the screen subroutine.

Which is very slow and roundabout. But this is the standard way of outputting characters that can be routed to DOS, a printer, the screen, or anywhere else you like. This process is extremely slow on the IIe when 80-column firmware is in use. So slow in fact, that you cannot keep up with a 1200-baud modem and scroll the screen at the same time.

The actual *screen subroutine* that puts the characters on the screen is called COUT1 and sits at $FDF0. Good old "Fideyfoo." Fideyfoo automatically keeps track of the horizontal and vertical character positions, does scrolls, handles carriage returns, inverses, your choice of flashing or lowercase, and takes care of most screen actions in the way that most people want most of the time.

Fideyfoo has some locations on page zero reserved that let you pick up special effects quickly and simply. For instance, the size of the scrolling window is set by locations $20 through $24. The cursor horizontal and vertical position bytes CH and CV are located at $24 and $25. Your choice of normal/inverse/flash is decided by INVFLG at $31. And the screen prompt is stashed in $33. See the EMPTY SHELL.SOURCE hooks for other locations of interest.

Here's how to remember when to use COUT or COUT1 . . .

Use COUT at $FDED to slowly output a character to DOS, a printer, the screen, or anywhere else you want to send that character. Hooks CSWL and CSWH at $36 and $37 decide where the character is to go.

Use COUT1 at $FDF0 to rapidly output a character only to the screen.

By the way, all these fancy subroutines do take time. It can take half a millisecond just to get through COUT and the DOS code, and any screen scrolls can hold up the works for four or more milliseconds. These times are on older Apples; the IIe is much worse in its 80-column mode. So, it pays to go directly to the screen or output device if speed is important.

It also pays to defeat any "screen echo" should you need top output speed. For instance, a HIRES graphics hard copy dump will be dramatically slowed down if it has to wait for screen scrolling on echo. For fastest possible speed, DOS could also be disconnected during character output times. And fast modems are best used on a IIe in its 40-column or "no-display" modes.

A third place to put your characters involves using a *HIRES character generator* to put your characters onto the HIRES screen. This type of subroutine lets you mix and match graphics and lets you use lots of different text fonts of varying sizes. You can also use special characters to do animation with a HIRES character generator, since your letter "G" is free to look like a frog's face, rather than a stock character. But HIRES character generators are usually rather slow and take bunches of extra code inside your machine.

Normally, a HIRES character generator will grab the COUT hooks "behind" DOS. Its use, once installed, will be pretty much the same as using COUT. Naturally, a HIRES character generator only will display on a HIRES screen and COUT1 will only display on a text screen.

A final place to put characters is to route them to your own custom code subroutine. This lets you rearrange things to suit yourself. A word processor is one example, where the messages all change from use to use. A second example could be a special effects screen filter. This one could "print" in oddball directions, and include delay, sound effects, replacements, screen locking, whole-word breaks, column justify, and most anything else you'd care to dream up.

Normally, you should avoid writing your own code if it more or less duplicates what is already available as ready-to-go subroutines in the Apple monitor. But special code can do special things special ways, and sometimes can give you a tremendous programming advantage over competitive programs.

An edge, even.

So, your first problem is to decide where your text is going to go.

Then, you have to pick some method of getting text to that destination.

Here are the names of several more popular text outputting methods, going from simple to complex . . .

TEXT OUTPUTTING SCHEMES
Brute Force
Short File
Long File
Imbedded Text
Compacted

The *brute force* method is simple and obvious. "Give me a D!" "Give me an O!" "Give me a G!" And whaddaya got? A doggedly cumbersome and very painful way to output text. Load the accumulator with the ASCII character for a D and then JSR your output code. Then load the accumulator with an ASCII "O," and so on.

This method is so painful, that you would only want to use it for a four-letter or shorter message, and then if that message was the only one in the program. Among other problems, note that five bytes of code are needed per character output.

The *short file* method is almost as obvious as the brute force method. Put your characters in a file. End each message with some marker, say an ASCII $00 or NUL. If you have to, create a second *pointer* file to tell you where each message starts. The short file is usually limited to 256 or fewer total characters.

The short file method uses indexed addressing to pick sequential characters out of a file. For a detailed example, see the text outenblatter in Volume II of *Don Lancaster's Micro Cookbook* (Sams 21829). Just for kicks, we will also use the short file method in the card shuffler of Ripoff Module 8.

The short file method is limited to a few very short and fixed text messages. But it is quick and simple to program, and may be all you need.

The *long file* method removes the 256 character restriction, by replacing 8-bit indexed loads with 16-bit indirect indexed loads. Your messages can now be any length and you can have any number of them, although there is a slight complication for more than 128 *different* messages at any time.

The long file method is more or less the "standard" way of handling medium length text messages, and is what this ripoff module is all about. We'll find out how your assembler can automate keeping track of messages and message pointers, as well as automatically entering ASCII characters for you.

But, there are limits to the long file method. All your messages must be known and all must be placed at one point in your code. The *imbedded text* method of the next ripoff module very elegantly gets around these restrictions, by letting you put any message you want, any place you want, *directly in your source code.* You can mix and match messages from any modules in your program, so long as one "un-imbedding" subroutine is provided somewhere in your code.

Both the long file and the imbedded text methods take around a byte per character for longer messages. You can remove the end marker on each string message if you switch from high ASCII to low ASCII on the last character. EDASM can do this for you as a special feature. But this complication doesn't save you very much, particularly on longer messages. You can also use a character count byte if you like. Again, this doesn't help much.

Should you have to really cram long messages into your Apple, you can either use repeated disk access or else use some *text compaction* scheme. Repeated disk access is very poor form these days and should be avoided, even with the newer DOS speedup tricks. Text compaction works by using some non-ASCII code that is more efficient than ASCII for character storage.

For instance, in the *Zork* adventures, three characters are crammed into two bytes, giving you code that needs only 67 percent of the space needed by ASCII. In the Adam's version of *Collossial Cave,* letters are arranged into pairs and then each pair is given an unique code. This results in nearly a 50 percent compaction. In spelling checkers, special codes are used to tell how many characters have not changed from the previous character. Special codes are also used for stock endings.

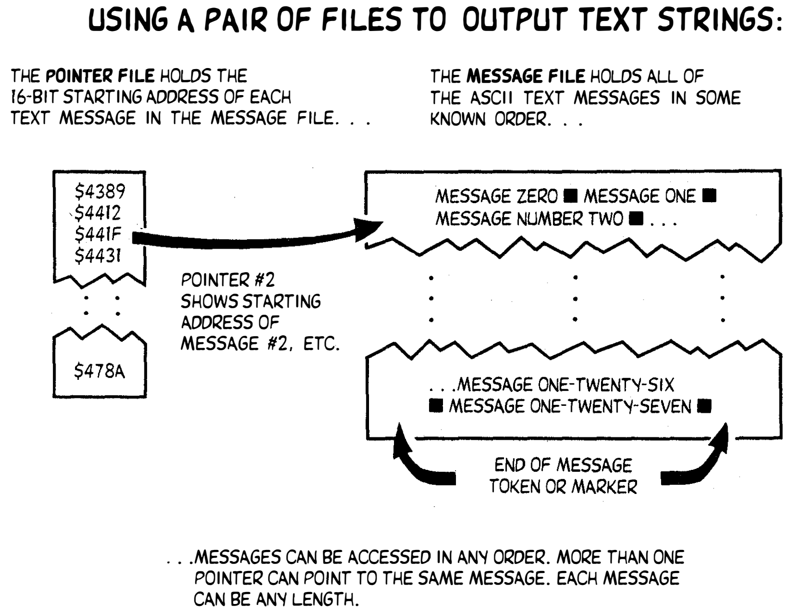
In general, you should not use text compaction until after you are sure you absolutely must have it. It's usually best to have your code completely debugged and your messages completely fixed before using compaction. Note that text compaction will actually *lengthen* and complicate the code needed for short messages, so there is some minimum "breakeven" code length before compaction gains you anything at all.

To recap, there are many places you can put characters and many different ways to generate text messages. One standard way is the text file method, which we will look at here. After that, in Ripoff Module 2, we will check into a more elegant imbedded text method that often is a better choice. Either of these methods is a good choice for your typical "medium" text message jobs, those not so trivial and short that you can handle with obvious code, nor those messages so long that you have to compact them.

So, without further ado, here is . . .

THE LONG FILE METHOD

There are two files involved in the long file method. One of these is called the *pointer file* and the other is called the *message file* . . .


USING A PAIR OF FILES TO OUTPUT TEXT STRINGS:
THE POINTER FILE HOLDS THE 16-BIT STARTING ADDRESS OF EACH TEXT MESSAGE IN THE MESSAGE FILE. . .
THE MESSAGE FILE HOLDS ALL OF THE ASCII TEXT MESSAGES IN SOME KNOWN ORDER. . .
$4389
$4412
$441F
$4431
$478A
POINTER #2 SHOWS STARTING ADDRESS OF MESSAGE #2, ETC.
MESSAGE ZERO ■ MESSAGE ONE ■
MESSAGE NUMBER TWO ■ . . .
. . .MESSAGE ONE-TWENTY-SIX
■ MESSAGE ONE-TWENTY-SEVEN ■
END OF MESSAGE TOKEN OR MARKER
. . .MESSAGES CAN BE ACCESSED IN ANY ORDER. MORE THAN ONE POINTER CAN POINT TO THE SAME MESSAGE. EACH MESSAGE CAN BE ANY LENGTH.

The long file method seems complicated at first, but this text outputting scheme lets you have messages of any length, and the messages can easily cross 256-byte page boundaries. You can also use different sets of pointer files and text files with the same FLPRINT subroutine.

The message file holds all the messages. The messages do not have to be in any particular order, but the order must be known. Each message ends with a marker of some sort. We will use an ASCII double zero NUL command, since it is easier to test for zero than for any other value. Normally, each message will follow the previous one, although this is not essential. Should you want to put a DOS message into your message file, you start the DOS message with a carriage return and a [D], or "<CTRL> D," otherwise known as an ASCII CR and EOT.

The pointer file holds a list of addresses that show the start of each message. Note that each pointer has to be a 16-bit, or two-byte, address, since the message file can be many pages long. Each pointer file is thus limited to 128 different message pointers, but you can have as many pointer files in your program as you like.

As you might guess, it can be a real drag building and connecting your files by hand. We will show you a fully automatic way to let your assembler build and link files for you. It's all done with creative use of labels.

To use the long file method, you first pick a pointer file. Then you decide which message you want. Say it is message 2. Then you read the pointer file to find the start of the message, say $441F. Then you reach into the message file, starting at $441F, get a character, and then output that character. You continue the process one character at a time till the marker comes up. Then you quit.

The messages do not have to be in any special order, and you can let several different pointers lead you to the same message. This gets handy for prompts like "Please make another selection;" or "That's not a letter, turkey!" defaults or error traps. You also can start at the *middle* of another message, and read to the end. This trick can sometimes save you space by using words over again.

The tricky part is being able to read long messages that cross page boundaries. To do this, you use the powerful 6502 indirect indexed command. In this ripoff module, we will set aside a pair of page zero address locations at $EB and $EC. When we decide to output a message, you reach into the pointer file and put the low half of the message starting address into $EB, and the high half of the message start into $EC.

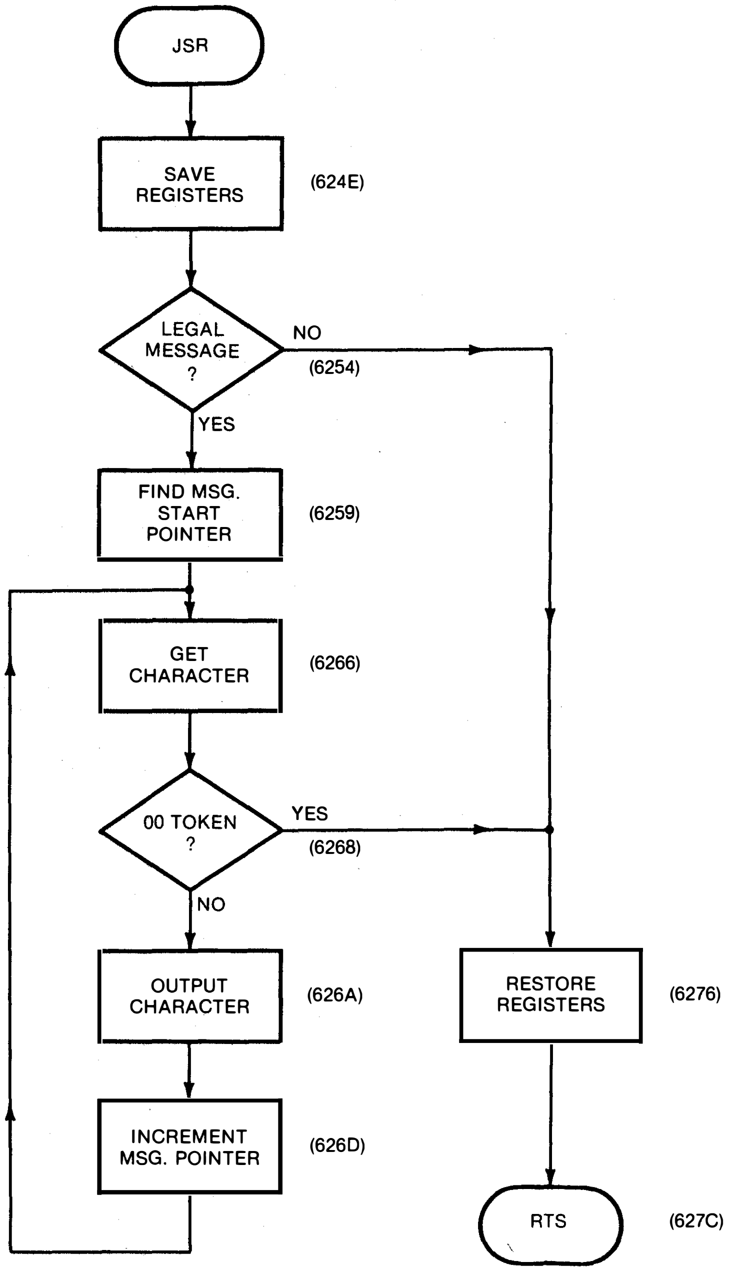
Then, you set your Y register to #00, and use the LDA($EB),Y indexed indirect addressing instruction. What this command does is go to the sum of the 16-bit address in $EB and $EC (the start of your message) *plus* the Y register value (zero) to get the character to be output.

After the first character, you have a choice. You could increment Y to get to the next character, or else you could add one to the $EB,EC pair. While adding one to Y seems faster and more attractive at first, this will only let you have 256 characters in any one message. So, we will keep Y at zero, and increment the base address. To increment a base address, you first increment the low byte at $EB. If you get a zero result, you then also increment the high byte at $EC. This way, you can continually work your way through most of the 64K address space, without any worries about page boundaries or running out of 8-bit range.

Holding the Y register at zero during an indexed indirect load simply "downgrades" the load command into a straight indirect load. Incidentally, the new 65C02s have "pure" or "unindexed" indirect commands that free up the Y register for other uses.

Confused? Here's a flowchart . . .

FLPRINT FLOWCHART:



Let's check into the actual code of the FLPRINT module, sitting at $624B. This is the module that outputs the text messages for you, and is what you will want to adapt to your own needs.

One good starting place to analyze any code is to find out where variables are stashed. On FLPRINT, we set aside two page zero locations at $EB and $EC to point to the start of our pointer file. These we call PFP1 and PFP1+1. We set aside two more locations at $ED and $EE to use as a running character pointer that works through the message file. These two are labeled MSP1 and MSP1+1.

We also provide a short stash at the end of the subroutine. Three locations here are used for a temporary Y register save YSAV1, an X register save XSAV1 and a total number-of-messages value at MNUM1.

You enter this FLPRINT module with the message number in the accumulator. You also must have pre-placed the pointer file starting address in PFP1.

We first save the X and Y registers into temporary stashes at XSAV1 and YSAV1. Next, you run a range check of the message number against the stash at MNUM1. A range check makes sure the message is a legal one. This keeps you from outputting garbage or plowing up a disk. We have used a MNUM1 value of $10, good for 16 separate messages.

If the message number is illegal, you restore the X and Y registers and exit without doing anything else. In a "real" program, you would error trap this and do something about it instead.

If the message number is valid, you double it with an ASL, since you are after *pairs* of addresses in the pointer file, each of which takes up 2 bytes.

Then, you reach into the pointer file and get the low half of the address and stash it at MSP1 and then grab the high half and dump it into MSP1+1. We knew *which* pointer file to go to, since whatever code that JSRed here put the pointer file starting address into PFP1 ahead of time.

At this point in the subroutine, we have placed an address into MSP1 and MSP1+1 that points to the first character in the desired message. Now, it's up to the service loop called NXTCHR1 to handle characters for us. NXTCHR1 first grabs a character. If that character was a double zero, the loop quits and exits via END1. This is how you end a message.

Usually, though, the character that NXTCHR1 grabs is not a double zero, so NXTCHR1 passes the character out to the Apple monitor subroutine at COUT that sends the character to whatever is connected to the output hooks.

Typically, the "hooked " character may go through DOS, which checks it for a [return] [D] header. If it doesn't look like something DOS is interested in, DOS then passes the character somewhere else, possibly to a printer card whose code may start at $C100. The printer card will send the character to a printer, and, optionally, will pass it on to the screen subroutine $FDF0 at COUT1.

None of which matters to NXTCHR1, for once this loop outputs a character to COUT, it couldn't care less what happens to the character. After the character is sent to wherever it is supposed to go, COUT returns control back to NXTCHR1 via a RTS subroutine return.

NXTCHR1's next job is to move the message file character pointer MSP1 over to the next character. Since this pointer is 16 bits wide, the low byte at MSP1 is first incremented. Should we get a zero result, indicating that a carry is needed to the high byte, we then increment the high byte. This is a pretty much standard way of incrementing a 16-bit address pointer pair.

Following that, we jump back to the start of the NXTCHR1 loop and keep outputting characters till we hit the double zero.

Note the forced branch at NOC1. It pays to keep absolute jumps out of any of your code modules, for absolute references make code harder to relocate. The CLC and BCC commands together do an unconditional relative branch for you that is easily relocatable.

After the double zero, we get out of the FLPRINT subroutine by restoring the X and Y registers and doing the usual RTS back to whoever it was that JSRed this module.

The DEMO1 that starts at $6200 is a rather unexciting "exerciser" that shows us how FLPRINT works. DEMO1 first sets the total number of messages to $03 and then finds out where the message pointer file sits. It then stores the pointer file start in MSP1, for use by FLPRINT.

Then we clear the screen, do some tabbing, and go to the inverse mode. Message #00 is called for, and gotten through FLPRINT. We return to normal text for message #01. Note that message #02 is quite long. It could, in fact, be any length you want, within the limits of available memory.

After message #01, we ask for user input. Should we get an "E," we exit the program. A "C" gives you a DOS catalog. This is done by printing first a CR and then an EOT, or [D], followed by the CATALOG string. If the CATALOG is long enough, extra prompts are needed for each catalog page.

Should you enter anything but an "E" or a "C," the entire FLPRINT module is rerun. This error trapping causes a brief flash on the screen, which should be enough of an operator "hey turkey!" prompt for most users.

In this example, we require a capital C or capital E. It is better practice to allow for either uppercase or lowercase entries. You can do this with a double test, or else by forcing lowercase characters into their uppercase equivalents. This important detail should not be omitted on the IIe or for older uses where you expect mixed cases. You'll find a case changer example in Ripoff Module 7.

Note that both the pointer file and the message file can be repeatedly reloaded off the disk. Thus, there is no limit to using external or calculated text strings as might be needed in a longer adventure.

Creating the Files

A good assembler will very much simplify setting up and creating your own pointer and text files. The process of putting the file into memory and properly linking it with everything else can be made fully automatic, without any worries about absolute addresses.

It's labels to the rescue.

Let's look at the message file first. First and foremost, you put a label at the beginning of each separate message. We have used M1.0 through M1.15 in the source code. If you have a label on your message, your assembler can find the message, regardless of where it ends up in memory.

We stopped at sixteen messages only to save on source code length. You can make things as long as you like, with up to 128 messages for each message pointer file and as many message pointer files as you want.

The ASC and DFB commands greatly simplify entering your messages. We've done almost everything here in uppercase for compatibility with older Apples, but most newer assemblers will let you use full case for your messages. "New way" editing also lets you do lowercase on most any assembler.

The ASC command tells the assembler to "convert what follows to ASCII." High ASCII is normally used in the Apple, although you can change this if you want to. While each ASCII string can be any length, it pays to keep each string under 32 characters or so. This makes for neater assembly listings. You can tie as many strings together as you need to get the total message.

A delimiter should start the ASCII text string. Use a quote for the delimiter unless you really want to print a quote. Then use a slash instead. An ending delimiter is not needed if there are no comments on the string line. Usually there won't be room for comments anyway, so this is no big deal. This is roughly similar to not needing the final quote in an Applesloth PRINT statement. Trailing spaces are hard to see without a final delimiter.

So much for alphanumerics. How do you handle control characters?

Obviously, you need a way to, say, imbed carriage returns. Yet if you type a carriage return, the string command completes itself. How do you get out of this bind?

Once again, it's labels to the rescue. Just as you can use a CHR$(13) to fool a higher level language into outputting a carriage return, you can trick an assembler into entering a carriage return into a file by using a label.

I've chosen to use single letter labels for control commands. B for backspace, C for carriage return, D for DOS, and X for the NULL or double zero. Each letter must be pre-defined as a constant, such as a B EQU $88 for a backspace. To enter control commands into your ASCII text, simply use DFBs with as many control commands as you like.

For instance, a DFB C,X puts a carriage return and an end-of-text marker into your message file. That wild DFB B,B,B,P,B,X sequence uses backspaces to center a flashing user prompt inside a fancy screen symbol. Incidentally, this may look different on a II and IIe. You might like to change it per the Apple in use.

Unfortunately, this was written before "new" EDASM became available. Since "A," "X," and "Y" are disallowed labels in "new" EDASM, you'll have to substitute something else for "X." Note that the STR pseudo-op in "new" EDASM can eliminate any need for a trailing NULL.

Summing up our message file, be sure to put a label on the start of each message. Then enter your ASCII characters using the assembler's ASC command. Don't forget the delimiter at the front and don't let the individual ASC strings get too long. Enter any control characters you want to imbed with DFB commands. On a typical message, you will alternate ASC and DFB commands. Use ASC for the letters and DFB for the carriage returns and end markers.

The pointer file will usually be much shorter than the message file. The pointer file holds the starting address of each message, so that FLPRINT knows where to go to start outputting characters.

To automate the construction of a pointer file, just use labels for each pointer entry. For instance, we call the pointer to the sixth message PF5 (don't forget that zero!). Our pointer source code under label PF5 tells us to DW M1.5, or to "go to wherever the message labeled M1.5 happens to be, find its present absolute address, and put that address pair back here."

Which is an awful lot of work for the assembly program. But that's its job and is one of the many reasons why we use an assembler in the first place—to automate most of the dogwork involved in writing machine language programs.

MIND BENDERS

—Show how FLPRINT can be simplified if you only have one pointer file in your program.

—FLPRINT works fine when called from *within* another program, but there's a slight bug when used directly from the monitor or Applesoft. What is the bug? What causes it? How can you prevent it?

—How can you design a program that outputs lowercase only to those machines that can use it?

—Can FLPRINT be used with changing messages? How?

—Show ways to use FLPRINT with several different pointer files.

—Rewrite this module to use "new" EDASM's string command STR, which includes a message count byte. What are the advantages of this new method?

PROGRAM RM-1
FILE BASED PRINTER

```
----- NEXT OBJECT FILE NAME IS FLPRINT
6200:              ORG  $6200     ; PUT MODULE #1 AT $6200


6200:           ;   *****************************************
6200:           ;   *                                       *
6200:           ;   *          -< FLPRINT MODULE >-         *
6200:           ;   *                                       *
6200:           ;   *       (FILE BASED STRING PRINTER)     *
6200:           ;   *                                       *
6200:           ;   *       VERSION 1.0  ($6200-$642A)      *
6200:           ;   *                                       *
6200:           ;   *               6-15-83                 *
6200:           ;   *.......................................*
6200:           ;   *                                       *
6200:           ;   *          COPYRIGHT C 1983 BY          *
6200:           ;   *                                       *
6200:           ;   *      DON LANCASTER AND SYNERGETICS    *
6200:           ;   *      BOX 1300, THATCHER AZ., 85552    *
6200:           ;   *                                       *
6200:           ;   *      ALL COMMERCIAL RIGHTS RESERVED   *
6200:           ;   *                                       *
6200:           ;   *****************************************


6200:           ;           *** WHAT IT DOES ***

6200:           ;   THIS MODULE OUTPUTS TEXT STRINGS OR DOS COMMANDS
6200:           ;   TO THE APPLE II'S OUTPUT HOOKS, USING STRINGS
6200:           ;   THAT ARE COLLECTED TOGETHER IN A COMMON FILE.
6200:           ;
6200:           ;
6200:           ;


6200:           ;           *** HOW TO USE IT ***

6200:           ;   YOUR CALLING CODE SHOULD HAVE PREVIOUSLY STORED
6200:           ;   A MESSAGE POINTER FILE ADDRESS IN PFP1 (LOW) AND
6200:           ;   PFP1+1 (HIGH).  ONE OF 128 POSSIBLE RESPONSES
6200:           ;   ARE SELECTED BY LOADING THE ACCUMULATOR WITH A
6200:           ;   MESSAGE NUMBER AND THEN DOING A JSR TO FLPRINT.
6200:           ;
```

PROGRAM RM-1, CONT'D . . .

```
6200:          ;             *** GOTCHAS ***

6200:          ;   THIS METHOD IS BEST USED FOR LONG MESSAGES THAT MIGHT
6200:          ;   NEED CALCULATED VALUES OR DISK-BASED CHANGES.
6200:          ;
6200:          ;   MESSAGES CAN BE ANY LENGTH, BUT MORE THAN 128 DIFFERENT
6200:          ;   MESSAGES WILL NEED SEPARATE MSP1 ADDRESS BASES.  EACH
6200:          ;   MESSAGE MUST END IN A $00 MARKER.


6200:          ;             *** ENHANCEMENTS ***

6200:          ;   DOS COMMANDS ARE OUTPUT BY STARTING THE STRING
6200:          ;   WITH A CARRIAGE RETURN AND <CTRL> D.
6200:          ;
6200:          ;   TO GO DIRECTLY TO THE SCREEN, USE COUT1 RATHER THAN COUT.
6200:          ;   THIS IS FASTER, BUT CANNOT CONTROL DOS OR BE PRINTED.
6200:          ;


6200:          ;             *** RANDOM COMMENTS ***

6200:          ;   TO RUN THE DEMO, USE $6200G OR CALL 25088.
6200:          ;
6200:          ;   THE X AND Y REGISTERS ARE PRESERVED; A IS DESTROYED.
6200:          ;
6200:          ;
6200:          ;
```

PROGRAM RM-1, CONT'D . . .

```
6200:            ;            *** HOOKS ***

FDED:              COUT     EQU  $FDED     ; OUTPUT CHARACTER VIA HOOKS
FC58:              HOME     EQU  $FC58     ; CLEAR SCREEN
FB2F:              INIT     EQU  $FB2F     ; INITIALIZE TEXT SCREEN
C010:              KBDSTR   EQU  $C010     ; KEYBOARD RESET
F94A:              PRBL2    EQU  $F94A     ; PRINT X BLANKS
FD0C:              RDKEY    EQU  $FD0C     ; GET INPUT CHARACTER
FE80:              SETINV   EQU  $FE80     ; SET INVERSE SCREEN
FE84:              SETNORM  EQU  $FE84     ; SET NORMAL SCREEN

00ED:              MSP1     EQU  $ED       ; MESSAGE FILE CHARACTER POINTER
00EB:              PFP1     EQU  $EB       ; POINTER FILE STARTING ADDRESS



6200:            ;            *** TEXTFILE COMMANDS ***

0088:              B        EQU  $88       ; BACKSPACE
008D:              C        EQU  $8D       ; CARRIAGE RETURN
0084:              D        EQU  $84       ; DOS ATTENTION
0060:              P        EQU  $60       ; FLASHING PROMPT
0000:              X        EQU  $00       ; END OF MESSAGE
```

PROGRAM RM-1, CONT'D . . .

```
6200:              ;           *** DEMO  ***
6200:              ;

6200:              ;      THE DEMO USES THE FLPRINT MODULE TO OUTPUT
6200:              ;      SCREEN MESSAGES AND A DOS CATALOG COMMAND.
6200:              ;

6200:A9 03           DEMO1    LDA  #$03      ; THREE MESSAGES TOTAL
6202:8D 7D 62                 STA  MNUM1     ; SAVE FOR CHECK
6205:A9 80                    LDA  #>PF0     ; SAVE MESSAGE POINTER LOW
6207:85 EB                    STA  PFP1      ;
6209:A9 62                    LDA  #<PF0     ; SAVE MESSAGE POINTER HIGH
620B:85 EC                    STA  PFP1+1    ;

620D:20 2F FB                 JSR  INIT      ; GO TO TEXT MODE
6210:20 58 FC                 JSR  HOME      ; CLEAR SCREEN
6213:A2 08                    LDX  #$08      ; PRINT BLANKS VIA MONITOR
6215:20 4A F9                 JSR  PRBL2     ;

6218:20 80 FE                 JSR  SETINV    ; INVERSE TEXT FOR TITLE
621B:A9 00                    LDA  #00       ; MESSAGE #0
621D:20 4E 62                 JSR  FLPRINT   ; PRINT MESSAGE

6220:20 84 FE                 JSR  SETNORM   ; NORMAL TEXT
6223:A9 01                    LDA  #$01      ; MESSAGE #1
6225:20 4E 62                 JSR  FLPRINT   ; PRINT MESSAGE

6228:2C 10 C0                 BIT  KBDSTR    ; RESET KEYBOARD
622B:20 0C FD                 JSR  RDKEY     ; GET KEY
622E:C9 C5                    CMP  #$C5      ; AN "E" FOR EXIT?
6230:F0 15                    BEQ  EXIT1     ;  YES, EXIT
6232:C9 C3                    CMP  #$C3      ; A "C" FOR CATALOG?
6234:D0 CA                    BNE  DEMO1     ; TRY AGAIN FOR VALID KEY

6236:20 58 FC                 JSR  HOME      ; CLEAR SCREEN, THEN
6239:A9 02                    LDA  #$02      ; DO CATALOG
623B:20 4E 62                 JSR  FLPRINT   ;
623E:2C 10 C0                 BIT  KBDSTR    ; HOLD CATALOG
6241:20 0C FD                 JSR  RDKEY     ;  TILL KEYPRESS

6244:18                       CLC            ; BRANCH ALWAYS
6245:90 B9                    BCC  DEMO1     ;  AND TRY AGAIN

6247:20 58 FC        EXIT1    JSR  HOME      ; EXIT DEMO
624A:2C 10 C0                 BIT  KBDSTR    ; RESET KEYBOARD
624D:60                       RTS            ;
```

PROGRAM RM-1, CONT'D . . .

```
624E:           ;             *** FLPRINT MODULE ***
624E:           ;
624E:           ;

624E:           ;           THIS MODULE USES THE ACCUMULATOR VALUE TO
624E:           ;           FIND A POINTER TO THE TEXT STRING. IT THEN
624E:           ;           OUTPUTS ONE CHARACTER AT A TIME TILL THE $00
624E:           ;           END-OF-MESSAGE MARKER IS FOUND.
624E:           ;
624E:           ;

624E:8C 7F 62   FLPRINT STY  YSAV1      ; SAVE REGISTERS
6251:8E 7E 62           STX  XSAV1      ;

6254:CD 7D 62           CMP  MNUM1      ; A LEGAL MESSAGE NUMBER?
6257:B0 1D              BCS  END1       ; DON'T PRINT IF ILLEGAL
6259:0A                 ASL  A          ; DOUBLE POINTER FOR ADDRESS PAIR
625A:A8                 TAY             ;
625B:B1 EB              LDA  (PFP1),Y   ; GET LOW POINTER
625D:85 ED              STA  MSP1       ;  AND SAVE
625F:C8                 INY             ;
6260:B1 EB              LDA  (PFP1),Y   ; GET HIGH POINTER
6262:85 EE              STA  MSP1+1     ;  AND SAVE
6264:A0 00              LDY  #$00       ; NO INDEXING
6266:B1 ED      NXTCHR1 LDA  (MSP1),Y   ; GET CHARACTER
6268:F0 0C              BEQ  END1       ; EXIT ON $00 MARKER
626A:20 ED FD           JSR  COUT       ; PRINT CHARACTER

626D:E6 ED              INC  MSP1       ; CALCULATE NEXT CHARACTER LOCATION
626F:D0 02              BNE  NOC1       ;   IF A CARRY, THEN
6271:E6 EE              INC  MSP1+1     ; INCREMENT HIGH ADDRESS LOCATION
6273:18         NOC1    CLC             ; BRANCH ALWAYS TO
6274:90 F0              BCC  NXTCHR1    ;  GET NEXT CHARACTER

6276:AE 7E 62   END1    LDX  XSAV1      ; RESTORE REGISTERS
6279:AC 7F 62           LDY  YSAV1      ;
627C:60                 RTS             ; AND EXIT



627D:           ;             *** STASH ***


627D:10         MNUM1   DFB  $10        ; NUMBER OF MESSAGES IN FILE
627E:00         XSAV1   DFD  $00        ; X-REGISTER SAVE
627F:00         YSAV1   DFB  $00        ; Y-REGISTER SAVE
```

PROGRAM RM-1, CONT'D . . .

```
6280:          ;             *** POINTER FILE ***

6280:A0 62     PF0      DW   M1.0       ; POINTER FILE
6282:B6 62     PF1      DW   M1.1       ;
6284:FA 63     PF2      DW   M1.2       ;
6286:05 64     PF3      DW   M1.3       ;
6288:08 64     PF4      DW   M1.4       ;
628A:0B 64     PF5      DW   M1.5       ;
628C:0E 64     PF6      DW   M1.6       ;
628E:11 64     PF7      DW   M1.7       ;
6290:14 64     PF8      DW   M1.8       ;
6292:17 64     PF9      DW   M1.9       ;
6294:1A 64     PF10     DW   M1.10      ;
6296:1D 64     PF11     DW   M1.11      ;
6298:20 64     PF12     DW   M1.12      ;
629A:23 64     PF13     DW   M1.13      ;
629C:26 64     PF14     DW   M1.14      ;
629E:29 64     PF15     DW   M1.15      ;



62A0:          ;             *** MESSAGE FILE ***

62A0:CD C5 D3  M1.0     ASC             "MESSAGE FILE METHOD"
62A3:D3 C1 C7
62A6:C5 A0 C6
62A9:C9 CC C5
62AC:A0 CD C5
62AF:D4 C8 CF
62B2:C4
62B3:8D 8D 00           DFB            C,C,X

62B6:D7 C9 D4  M1.1     ASC            "WITH THIS METHOD,  ALL OF THE MESSAGES
62B9:C8 A0 D4
62BC:C8 C9 D3
62BF:A0 CD C5
62C2:D4 C8 CF
62C5:C4 AC A0
62C8:A0 C1 CC
62CB:CC A0 CF
62CE:C6 A0 D4
62D1:C8 C5 A0
62D4:CD C5 D3
62D7:D3 C1 C7
62DA:C5 D3 A0
62DD:8D                 DFB             C
```

PROGRAM RM-1, CONT'D . . .

```
62DE:C1 D2 C5  217          ASC              "ARE COMBINED INTO A COMMON TEXT FILE
62E1:A0 C3 CF
62E4:CD C2 C9
62E7:CE C5 C4
62EA:A0 C9 CE
62ED:D4 CF A0
62F0:C1 A0 C3
62F3:CF CD CD
62F6:CF CE A0
62F9:D4 C5 D8
62FC:D4 A0 C6
62FF:C9 CC C5
6302:AE
6303:8D 8D     218          DFB              C,C

6305:C1 A0 D0  220          ASC              "A POINTER FILE IS USED TO DECIDE WHICH
6308:CF C9 CE
630B:D4 C5 D2
630E:A0 C6 C9
6311:CC C5 A0
6314:C9 D3 A0
6317:D5 D3 C5
631A:C4 A0 D4
631D:CF A0 C4
6320:C5 C3 C9
6323:C4 C5 A0
6326:D7 C8 C9
6329:C3 C8 A0
632C:8D        221          DFB              C

632D:CD C5 D3  223          ASC              "MESSAGE IS TO BE OUTPUT.
6330:D3 C1 C7
6333:C5 A0 C9
6336:D3 A0 D4
6339:CF A0 C2
633C:C5 A0 CF
633F:D5 D4 D0
6342:D5 D4 AE
6345:8D 8D     224          DFB              C,C

6347:D5 D3 C5  226          ASC              "USES INCLUDE TEXT DATA BASES AND OTHER
634A:D3 A0 C9
634D:CE C3 CC
6350:D5 C4 C5
6353:A0 D4 C5
6356:D8 D4 A0
6359:C4 C1 D4
635C:C1 A0 C2
635F:C1 D3 C5
6362:D3 A0 C1
6365:CE C4 A0
6368:CF D4 C8
636B:C5 D2
636D:8D        227          DFB              C
```

PROGRAM RM-1, CONT'D . . .

```
636E:D0 CC C1  230       ASC        "PLACES WHERE LOTS OF CHANGING MESSAGE
6371:C3 C5 D3
6374:A0 D7 C8
6377:C5 D2 C5
637A:A0 CC CF
637D:D4 D3 A0
6380:CF C6 A0
6383:C3 C8 C1
6386:CE C7 C9
6389:CE C7 A0
638C:CD C5 D3
638F:D3 C1 C7
6392:C5 D3
6394:8D        231       DFB        C

6395:C1 D2 C5  233       ASC        "ARE TO BE PRINTED OR DISPLAYED.
6398:A0 D4 CF
639B:A0 C2 C5
639E:A0 D0 D2
63A1:C9 CE D4
63A4:C5 C4 A0
63A7:CF D2 A0
63AA:C4 C9 D3
63AD:D0 CC C1
63B0:D9 C5 C4
63B3:AE
63B4:8D 8D 8D  234       DFB        C,C,C,C
63B7:8D
63B8:D4 D9 D0  235       ASC        /TYPE "C" FOR CATALOG, OR "E" FOR EXIT
63BB:C5 A0 A2
63BE:C3 A2 A0
63C1:C6 CF D2
63C4:A0 C3 C1
63C7:D4 C1 CC
63CA:CF C7 AC
63CD:A0 CF D2
63D0:A0 A2 C5
63D3:A2 A0 C6
63D6:CF D2 A0
63D9:C5 D8 C9
63DC:D4 AE
63DE:8D 8D 8D  236       DFB        C,C,C,C
63E1:8D
63E2:A0 A0 A0  237       ASC        "             -< >-"
63E5:A0 A0 A0
63E8:A0 A0 A0
63EB:A0 A0 A0
63EE:A0 AD BC
63F1:A0 BE AD
63F4:88 88 88  238       DFB        B,B,B,P,B,X
63F7:60 88 00
```

PROGRAM RM-1, CONT'D . . .

```
63FA:8D 84      241 M1.2    DFB             C,D
63FC:C3 C1 D4   242         ASC             "CATALOG"
63FF:C1 CC CF
6402:C7
6403:8D 00      243         DFB             C,X

6405:A0         245 M1.3    ASC             " "
6406:8D 00      246         DFB             C,X

6408:A0         248 M1.4    ASC             " "
6409:8D 00      249         DFB             C,X

640B:A0         251 M1.5    ASC             " "
640C:8D 00      252         DFB             C,X

640E:A0         254 M1.6    ASC             " "
640F:8D 00      255         DFB             C,X

6411:A0         257 M1.7    ASC             " "
6412:8D 00      258         DFB             C,X

6414:A0         260 M1.8    ASC             " "
6415:8D 00      261         DFB             C,X

6417:A0         263 M1.9    ASC             " "
6418:8D 00      264         DFB             C,X

641A:A0         266 M1.10   ASC             " "
641B:8D 00      267         DFB             C,X

641D:A0         269 M1.11   ASC             " "
641E:8D 00      270         DFB             C,X

6420:A0         272 M1.12   ASC             " "
6421:8D 00      273         DFB             C,X

6423:A0         275 M1.13   ASC             " "
6424:8D 00      276         DFB             C,X

6426:A0         278 M1.14   ASC             " "
6427:8D 00      279         DFB             C,X

6429:A0         281 M1.15   ASC             " "
642A:8D 00      282         DFB             C,X



*** SUCCESSFUL ASSEMBLY: NO ERRORS
```

2

IMBEDDED STRING PRINTER

a powerful and very sneaky way of mixing and matching text messages

I guess I've always been attracted to elegant simplicity, particularly when it is combined with sneakiness. The file based text printer of Ripoff Module 1 is a classic and standard old warhorse that's cumbersome, restrictive, and hard to use. It obviously doesn't qualify. What can we do that is better?

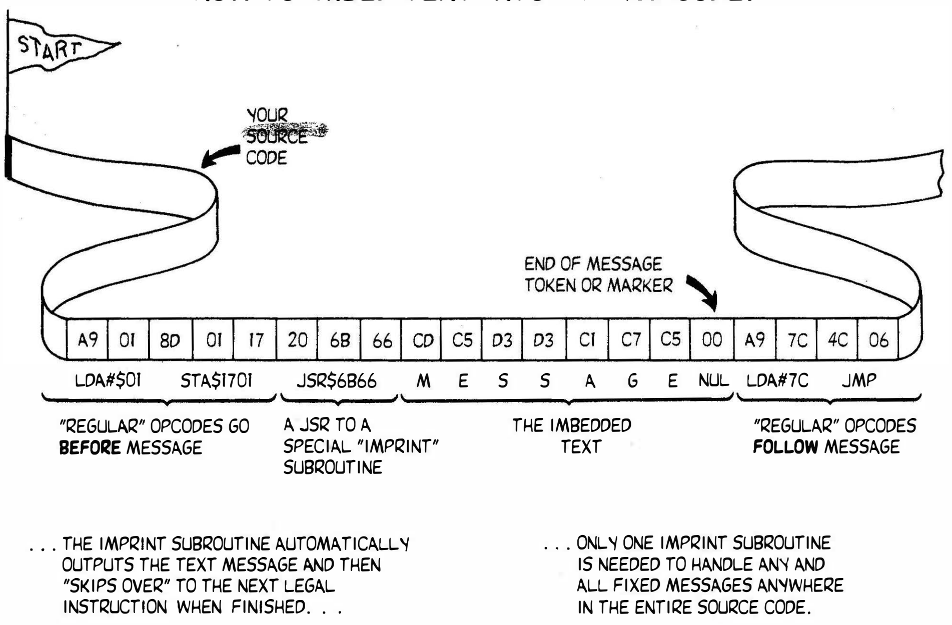
Why do we need a text message file at all? Why not, instead, simply *imbed* the text messages *directly into the* ~~*source*~~ *code* when and where they are needed? This way, you can have any number of short and fixed messages anywhere in your program, and you can mix and match modules from all over the lot without any worries at all about creating a big master text file and bunches of pointers to work with it.

The usual excuse for not imbedding text into ~~source~~ code is that the 6502 tends to get violently ill when you feed it ASCII text instead of machine language commands. The trick is to find some elegantly simple way to keep the imbedded messages out of the CPU. The way is called the *imbedded text* method.

With the imbedded text method, you simply insert ASCII text or DOS strings into your ~~source~~ code when and as you need them. Immediately before the strings, you do a jump to a very special subroutine that will grab all the ASCII stuff for its own use, and then let the 6502 pick up the machine language commands that *follow* the message.

Like so . . .

HOW TO IMBED TEXT INTO SOURCE CODE:


START
YOUR SOURCE CODE
END OF MESSAGE TOKEN OR MARKER
A9 01 8D 01 17 20 6B 66 CD C5 D3 D3 C1 C7 C5 00 A9 7C 4C 06
LDA#$01 STA$1701 JSR$6B66 M E S S A G E NUL LDA#7C JMP
"REGULAR" OPCODES GO BEFORE MESSAGE
A JSR TO A SPECIAL "IMPRINT" SUBROUTINE
THE IMBEDDED TEXT
"REGULAR" OPCODES FOLLOW MESSAGE
. . . THE IMPRINT SUBROUTINE AUTOMATICALLY OUTPUTS THE TEXT MESSAGE AND THEN "SKIPS OVER" TO THE NEXT LEGAL INSTRUCTION WHEN FINISHED. . .
. . . ONLY ONE IMPRINT SUBROUTINE IS NEEDED TO HANDLE ANY AND ALL FIXED MESSAGES ANYWHERE IN THE ENTIRE SOURCE CODE.

You will need only one imbedded printing subroutine. This can go anywhere in your program. That sub is called IMPRINT. Any and all program modules can use this lone IMPRINT subroutine any time they want to output a fixed text message. While most of these messages will usually be short, there is essentially no limit, except for memory space, as to how long your messages are, how many messages you use, or how you mix and match them

And all this *without* any pointers or master text files.

IMPRINT works by first finding out who called it. It does this by looking into the stack to find the intended subroutine return address. Not only does IMPRINT find the return address, but it *steals* it off the stack and uses that address as a string pointer. It then increments the return address ASCII character by ASCII character, until the message is finished. Finally, IMPRINT forces a subroutine return that goes *beyond* the imbedded text and picks up on the next mainstream machine language command.

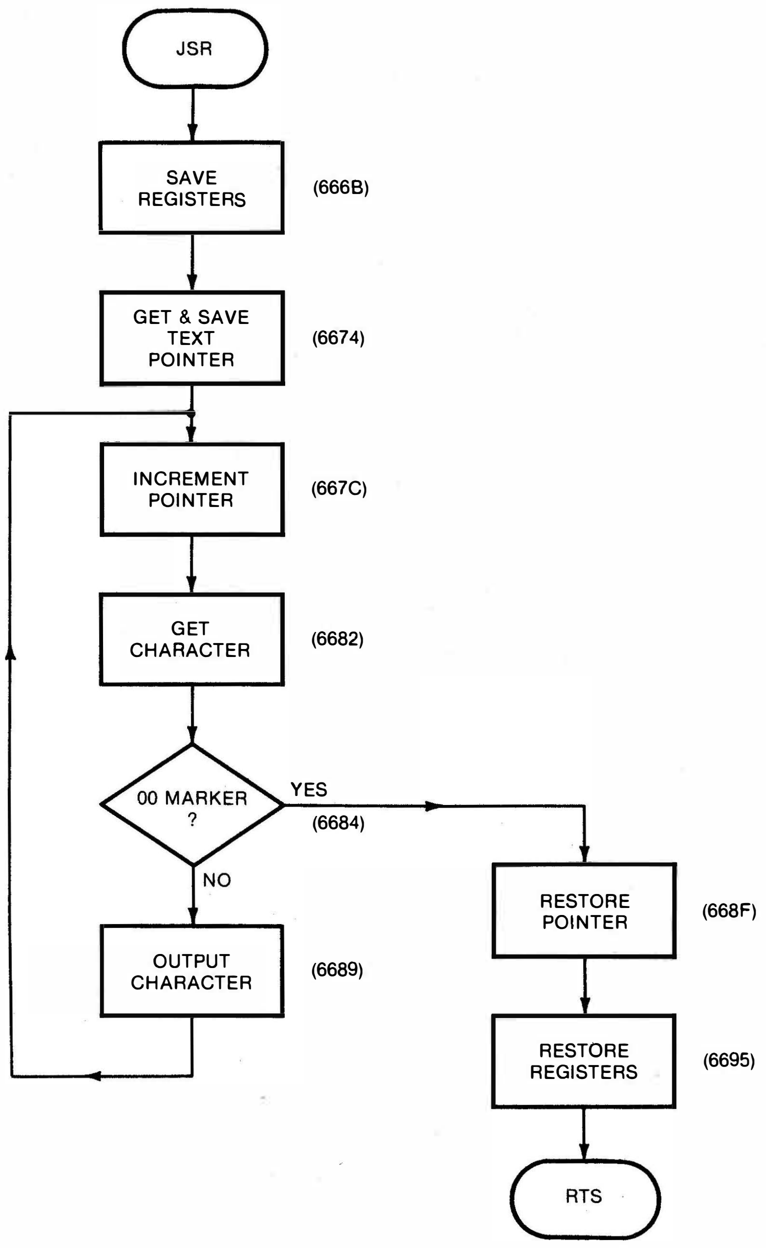
What is elegant and sneaky about the whole thing is that IMPRINT is not really a subroutine at all! IMPRINT is a "mainline" code module that "plugs itself" into high level code when and where it is called. It does this by messing with the stack. First, it pulls the return address off the stack, converting itself into mainline code. When finished outputting text, IMPRINT pushes the return-to-the-next-machine-language address onto the stack and then does a quick RTS, which is nothing but a forced jump.

Sounds hairy.

And it is. But the code is very short and simple. It's also very easy to use once you understand it. And, as further elegance, IMPRINT does not hurt *any* working registers at all.

Here's a flowchart of IMPRINT . . .

IMPRINT FLOWCHART:



IMPRINT sits at $666B right now, but it is easily put any place you want.

As before, to understand a machine language module, find out what variables are stashed where. Two slots on page zero are set aside as a pointer to the character being output. These are called STRP2 and STRP2+1 and are located at $EB and $EC. Three absolute slots are used to save the registers, and are called ASAV2, XSAV2, and YSAV2, and appear as a short stash that follows IMPRINT.

An aside or two. A pair of mnemonics involved in a 16-bit word can be spelled out either as STRP2L and STRP2H, or as STRP2 and STRP2+1. The H and L stand for high and low. The standard way is to use high and low, but you save on code and EQUs by using the arithmetic addition feature of your assembler. Do an EQU on STRP2, and your STRP2+1 rides along free.

By the way, the 2 tag just stands for module 2. This way, you can combine the ripoff modules anyway you like without worrying about duplicate label errors for common names.

Secondly, there are many different ways to temporarily save your accumulator and X- and Y-registers. It is usually a good idea to save all working registers during a subroutine or service module, so you keep any surprises out of the calling code. We have used absolute stores, since they are the safest and surest way of stashing things without memory conflicts. Absolute stores can take more bytes, can be slower, and are a somewhat harder to relocate than other storage methods. Page zero stores are faster, but you tie up precious and possibly conflicting real estate when you try this. The stack is another obvious stash, but its use gets messy fast, particularly on code like IMPRINT that purposely messes with the stack.

The absolute *worst* place to save working registers is in the monitor register saving subroutines IOSAVE and IOREST . . .

Don't EVER use the monitor routines IOSAVE and IOREST!

Sooner or later, they are bound to create problems.

What happens is that some module will use IOSAVE for its register saves and then may JSR to some other module that also tries to use IOSAVE for its own use. The first save gets overwritten by the second, and the final IOREST does a self-destruct, rather than a restore.

Let's see. Where were we? Back to IMPRINT. We first save our registers to the three absolute locations ASAV2, XSAV2, and YSAV2, and stash these at the end of the module.

The subroutine return address in the 6502's stack pointer takes two bytes. The *low address* is the first one you get back. The *high address* is the second byte you get back. That address points to *one less* than where you end up . . .

6502 SUBROUTINE STACK RULES

Two bytes on a stack are used to save a subroutine return address.

The FIRST byte you get back holds the return address POSITION byte.

The SECOND byte you get back holds the return address PAGE byte.

The RTS command returns you to the return address PLUS ONE.

So, we grab the top of the stack and store it as the low address half at STRP2. Then we grab the top of the stack again, and this time store it as the high address half at STRP2+1.

But, note at this time that this "return" address is pointing to *one less than* our first ASCII character, rather than to a "safe" 6502 return point. Note also that *we are no longer in a subroutine.* Why? Because the calling code pushed two things onto the stack, and the using code pulled two things back off of the stack. We are thus once again back in high level code!

To get our string pointer STRP2 pointing to our first ASCII character, we simply increment the pair in the usual way. Do this by incrementing STRP2, and then, if you get a zero result, take care of the overflow by incrementing STRP2+1. Since we know we will have to increment to get between characters, we'll arrange things so we only need one increment command, at the head of the loop called NXTCHH2.

Your ASCII or DOS text string gets entered into your calling source code, and should end with some marker. We will use the ASCII double zero NULL command here, since it is simplest.

At this time, we grab the character from the string using the indirect indexed loading that lets us reach any point in the 16-bit address space without any page boundary worries. As before, we have forced the Y-register to $00, to downgrade the indirect indexed command into a "pure" indirect load.

Having gotten the character, we can test it for a double zero. If we get the double zero, we go on to the exit routine at END2. If not, we output the character to COUT or to Fideyfoo, or wherever.

Next, we have included a JSR to an immediate return that we call HOOK2. This has no present use, but it lets you grab IMPRINT for special effects such as character delay, sound, printing in a weird screen direction, or whatever. To use it, just let the subroutine lead you to your special effects module.

After this unused hook, a relative forced branch that fakes an unconditional jump gets us back to NXTCHR2 and completes the loop.

Processing continues one character at a time until we get to the double zero. Then we branch down to the END2 routine.

At this time, the STRP2 pointer is pointing to the double zero of the last character, which is one less than the address of the continuing machine language code in the mainstream. On a subroutine return, the RTS command always goes to one more than the return address. So, *STRP2 equals the correct subroutine return address* when it is pointing to the end-of-text marker.

All the remains is to get back to the mainstream code. We might be tempted to try using the jump indirect instruction, but this one has a deadly bug that will nail you one time out of 128 . . .

The JMP indirect command has a deadly bug in it that misses page boundary crossings.

DON'T USE IT!

The newer 65C02's have fixed this bug, but they are not yet in wide use at this writing.

We will return to the main code by the exact opposite way we got into IMPRINT. First we shove the high half of the return address minus one, or STRP2+1 onto the stack, and then we shove the low half of the return address minus one, or STRP2, onto the stack. Miraculously, we are now back into a subroutine. To exit, you simply do a RTS.

On the subroutine return, you return to your mainstream code, exactly on the first valid instruction *following* your text message. Very nicely, all the text went out by way of IMPRINT, and the 6502 is ready to continue on the first valid instruction that follows the message.

Note carefully what happened. We go merrily along doing the usual op codes in the usual way. Then we JSR to some very special code that reads and then outputs everything that follows as text. This continues until an end marker. Then, the special code automatically ''skips over'' the text part, letting you pick back up on the conventional op codes that follow.

At no time does the 6502 see anything but legal op codes. While there is a big ''hole'' in your source code that holds text, this part of your source code never gets to the CPU. Nifty.

A Demo

To use IMPRINT, just load it into a known location in your Apple. In any module where you want to output a text message, insert a JSR IMPRINT, followed by the message, followed by a double zero marker. Then pick up your continuing code, just like you normally would.

DEMO2 shows us how it's done. We first initialize to the text mode, clear the screen, do a tab to center a title, and then switch to inverse. Next, our first message is put down by JSRing to IMPRINT, followed by the ''Imbedded String Method'' title. We then go back to normal text for a few lines, followed by an inverse ''JSR,'' and more normal text. The messages can be combined end on end as shown. This lets you have long messages that will still print neatly on your source code listing. Once again, the assembler enters the character strings with an ASC command, and enters control commands and end markers using DFBs.

Lines 155 and 156 show how to put a prompt into a fancy cue box. You might want to modify this slightly for best IIe results.

As with the file printer, a DOS command is done by starting with a CR and EOT, or [D] followed by a legal DOS instruction. The user can pick an ''E'' for exit or a ''C'' for catalog to demonstrate DOS access. Any other key reprints the message, giving a subtle, obvious, and non-obnoxious cue to the user that he is not paying attention.

The imbedded string method is far better than the file based text printer and the previous ripoff module, particularly when lots of fixed and fairly short messages are spread out in a mix-and-match fashion from program module to module.

Elegant simplicity.

MIND BENDERS

—Show how the IMPRINT method can be used with changing, calculated, or disk-based text.

—What else can you do with the concept of a JSR, followed by parameters or values needed by that sub, imbedded in mainstream code?

—Are there any advantages to using BRK to call IMPRINT? How would you do this? What are the limitations?

—Show how "new" EDASM's byte-counting LST pseudo-op can improve this module.

—How can you link an assembler with a word processor so that long text messages can be easily edited and entered into source code?

—Under what circumstances would you NOT want to use IMPRINT?

PROGRAM RM-2
IMBEDDED STRING PRINTER

```
----- NEXT OBJECT FILE NAME IS IMPRINT
6500:              ORG  $6500      ; PUT MODULE #2 AT $6500


6500:         ;   ****************************************
6500:         ;   *                                      *
6500:         ;   *        -< IMPRINT MODULE >-          *
6500:         ;   *                                      *
6500:         ;   *      (IMBEDDED STRING PRINTER)       *
6500:         ;   *                                      *
6500:         ;   *     VERSION 1.0  ($6500-$66A1)       *
6500:         ;   *                                      *
6500:         ;   *              6-15-83                 *
6500:         ;   *......................................*
6500:         ;   *                                      *
6500:         ;   *         COPYRIGHT C 1983 BY          *
6500:         ;   *                                      *
6500:         ;   *    DON LANCASTER AND SYNERGETICS     *
6500:         ;   *    BOX 1300, THATCHER AZ., 85552     *
6500:         ;   *                                      *
6500:         ;   *    ALL COMMERCIAL RIGHTS RESERVED    *
6500:         ;   *                                      *
6500:         ;   ****************************************


6500:         ;           *** WHAT IT DOES ***

6500:         ;   THIS MODULE OUTPUTS TEXT STRINGS OR DOS COMMANDS
6500:         ;   TO THE APPLE II'S OUTPUT HOOKS, USING STRINGS
6500:         ;   THAT ARE DIRECTLY IMBEDDED IN THE SOURCE CODE.
6500:         ;
6500:         ;
6500:         ;


6500:         ;           *** HOW TO USE IT ***

6500:         ;   YOUR CALLING CODE SHOULD HAVE A JSR TO IMPRINT.
6500:         ;   THIS JSR SHOULD BE IMMEDIATELY FOLLOWED BY AN ASCII
6500:         ;   STRING ENDING WITH AN $00 MARKER.
6500:         ;
6500:         ;
6500:         ;
```

PROGRAM RM-2, CONT'D . . .

```
6500:           ;           *** GOTCHAS ***

6500:           ;   THIS METHOD IS BEST USED FOR SHORT, UNRELATED MESSAGES
6500:           ;   INTERNAL TO YOUR PROGRAM.
6500:           ;
6500:           ;   MESSAGES CAN BE ANY LENGTH, BUT MORE THAN 40 CHARACTERS
6500:           ;   WILL NOT PRINT CLEANLY ON THE ASSEMBLY LISTING.
6500:           ;


6500:           ;          *** ENHANCEMENTS ***

6500:           ;   DOS COMMANDS ARE OUTPUT BY STARTING THE STRING
6500:           ;   WITH A CARRIAGE RETURN AND <CTRL> D.
6500:           ;
6500:           ;   TO GO DIRECTLY TO THE SCREEN, USE COUT1 RATHER THAN COUT.
6500:           ;   THIS IS FASTER, BUT CANNOT CONTROL DOS OR BE PRINTED.
6500:           ;


6500:           ;          *** RANDOM COMMENTS ***

6500:           ;   TO RUN THE DEMO, USE $6500G OR CALL 25856.
6500:           ;
6500:           ;
6500:           ;
6500:           ;
6500:           ;
```

PROGRAM RM-2, CONT'D . . .

```
6500:         ;             *** HOOKS ***

FDED:         COUT     EQU  $FDED      ; OUTPUT CHARACTER VIA HOOKS
FC58:         HOME     EQU  $FC58      ; CLEAR SCREEN
C010:         KBDSTR   EQU  $C010      ; KEYBOARD RESET
FB2F:         INIT     EQU  $FB2F      ; INITIALIZE TEXT SCREEN
FD1B:         KEYIN    EQU  $FD1B      ; READ KEYBOARD
F94A:         PRBL2    EQU  $F94A      ; PRINT X BLANKS
FE80:         SETINV   EQU  $FE80      ; SET INVERSE SCREEN
FE84:         SETNORM  EQU  $FE84      ; SET NORMAL SCREEN
FCA8:         WAIT     EQU  $FCA8      ; TIME DELAY SET BY ACCUMULATOR

00EB:         STRP2    EQU  $EB        ; POINTER TO ASCII STRING



6500:         ;            *** TEXTFILE COMMANDS ***

0088:         B        EQU  $88        ; BACKSPACE
008D:         C        EQU  $8D        ; CARRIAGE RETURN
0084:         D        EQU  $84        ; DOS ATTENTION
008A:         L        EQU  $8A        ; LINEFEED
0060:         P        EQU  $60        ; FLASHING PROMPT
0000:         X        EQU  $00        ; END OF MESSAGE
```

PROGRAM RM-2, CONT'D . . .

```
6500:              ;          *** DEMO  ***
6500:              ;
6500:              ;

6500:              ;        THE DEMO USES THE IMPRINT MODULE TO OUTPUT
6500:              ;        SCREEN MESSAGES AND A DOS CATALOG COMMAND.
6500:              ;
6500:              ;
6500:              ;
6500:              ;

6500:20 2F FB  DEMO2    JSR  INIT       ; GO TO TEXT MODE
6503:20 58 FC           JSR  HOME       ; CLEAR SCREEN
6506:A2 07              LDX  #07        ; ADD BLANKS TO START
6508:20 4A F9           JSR  PRBL2      ;
650B:20 80 FE           JSR  SETINV     ; INVERSE HEADER
650E:20 6B 66           JSR  IMPRINT    ; PUT DOWN HEADER

6511:8A 8A 8A           DFB              L,L,L
6514:C9 CD C2           ASC              "IMBEDDED STRING METHOD"
6517:C5 C4 C4
651A:C5 C4 A0
651D:D3 D4 D2
6520:C9 CE C7
6523:A0 CD C5
6526:D4 C8 CF
6529:C4
652A:8D 8D 00           DFB              C,C,X

652D:20 84 FE           JSR  SETNORM    ; NORMAL TEXT
6530:20 6B 66           JSR  IMPRINT    ; TOP TEXT LINE

6533:D7 C9 D4           ASC             "WITH THIS METHOD, EACH MESSAGE STRING
6536:C8 A0 D4
6539:C8 C9 D3
653C:A0 CD C5
653F:D4 C8 CF
6542:C4 AC A0
6545:C5 C1 C3
6548:C8 A0 CD
654B:C5 D3 D3
654E:C1 C7 C5
6551:A0 D3 D4
6554:D2 C9 CE
6557:C7 A0
6559:8D                 DFB              C
```

PROGRAM RM-2, CONT'D . . .

```
655A:C6 CF CC         ASC             "FOLLOWS ITS OWN
655D:CC CF D7
6560:D3 A0 C9
6563:D4 D3 A0
6566:CF D7 CE
6569:A0
656A:00               DFB              X

656B:20 80 FE         JSR  SETINV     ; INVERSE TEXT
656E:20 6B 66         JSR  IMPRINT    ;

6571:CA D3 D2         ASC             "JSR"
6574:00               DFB              X

6575:20 84 FE         JSR  SETNORM    ; RETURN TO NORMAL TEXT
6578:20 6B 66         JSR  IMPRINT    ;  AFTER JSR

657B:A0 C3 C1         ASC             " CALL, IMBEDDED IN
657E:CC CC AC
6581:A0 C9 CD
6584:C2 C5 C4
6587:C4 C5 C4
658A:A0 C9 CE
658D:8D               DFB              C

658E:C9 D4 D3         ASC              "ITS OWN SOURCE CODE.  "
6591:A0 CF D7
6594:CE A0 D3
6597:CF D5 D2
659A:C3 C5 A0
659D:C3 CF C4
65A0:C5 AE A0
65A3:A0

65A4:CE CF A0         ASC              "NO POINTERS AND
65A7:D0 CF C9
65AA:CE D4 C5
65AD:D2 D3 A0
65B0:C1 CE C4
65B3:8D               DFB              C

65B4:CE CF A0         ASC              "NO MASTER FILE ARE NEEDED.
65B7:CD C1 D3
65BA:D4 C5 D2
65BD:A0 C6 C9
65C0:CC C5 A0
65C3:C1 D2 C5
65C6:A0 CE C5
65C9:C5 C4 C5
65CC:C4 AE
65CE:8D 8D            DFB              C,C
```

PROGRAM RM-2, CONT'D . . .

```
65D0:C2 C5 D3  149         ASC              "BEST USE IS FOR FIXED, SHORT MESSAGES.
65D3:D4 A0 D5
65D6:D3 C5 A0
65D9:C9 D3 A0
65DC:C6 CF D2
65DF:A0 C6 C9
65E2:D8 C5 C4
65E5:AC A0 D3
65E8:C8 CF D2
65EB:D4 A0 CD
65EE:C5 D3 D3
65F1:C1 C7 C5
65F4:D3 AE
65F6:8D 8D     150         DFB               C,C

65F8:D4 D9 D0  152         ASC              /TYPE "C" FOR CATALOG, OR "E" FOR EXIT.
65FB:C5 A0 A2
65FE:C3 A2 A0
6601:C6 CF D2
6604:A0 C3 C1
6607:D4 C1 CC
660A:CF C7 AC
660D:A0 CF D2
6610:A0 A2 C5
6613:A2 A0 C6
6616:CF D2 A0
6619:C5 D8 C9
661C:D4 AE
661E:8D 8D     153         DFB               C,C

6620:A0 A0 A0  155         ASC              "                 -< >-"
6623:A0 A0 A0
6626:A0 A0 A0
6629:A0 A0 A0
662C:A0 A0 AD
662F:BC A0 BE
6632:AD
6633:88 88 88  156         DFB               B,B,B,P,B,X
6636:60 88 00
```

PROGRAM RM-2, CONT'D . . .

```
6639:2C 10 C0  AGAIN2  BIT  KBDSTR     ; RESET KEY STROBE
663C:20 1B FD  KBD2    JSR  KEYIN      ; READ KEYBOARD
663F:C9 C5             CMP  #$C5       ; AN "E" FOR EXIT?
6641:F0 21             BEQ  EXIT2      ; YES, EXIT
6643:C9 C3             CMP  #$C3       ; A "C" FOR CATALOG?
6645:D0 1A             BNE  RETRY2     ; NO, REPRINT SCREEN

6647:20 58 FC          JSR  HOME       ; CLEAR SCREEN FOR CATALOG
664A:20 6B 66          JSR  IMPRINT    ;

664D:8D 84             DFB  C,D        ; DOS HEADER
664F:C3 C1 D4          ASC  "CATALOG"  ;
6652:C1 CC CF
6655:C7
6656:8D 00             DFB  C,X        ; DOS TRAILER

6658:20 6B 66          JSR  IMPRINT    ; PROMPT AFTER CATALOG
665B:8D 60 00          DFB              C,P,X


665E:18                CLC             ; BRANCH ALWAYS
665F:90 D8             BCC  AGAIN2     ;
6661:4C 00 65  RETRY2  JMP  DEMO2      ; TOO FAR FOR BRANCH
6664:20 58 FC  EXIT2   JSR  HOME       ; CLEAR SCREEN
6667:2C 10 C0          BIT  KBDSTR     ; RESET KEYSTROBE
666A:60                RTS             ;  AND RETURN
```

PROGRAM RM-2, CONT'D . . .

```
666B:          ;             *** IMPRINT MODULE ***
666B:          ;
666B:          ;


666B:          ;           THIS MODULE UNPOPS THE STACK TO FIND THE
666B:          ;           IMBEDDED STRING.  IT OUTPUTS ONE CHARACTER
666B:          ;           AT A TIME TILL A $00 MARKER IS FOUND. THEN
666B:          ;           IT JUMPS BACK TO THE CALLING PROGRAM JUST
666B:          ;           BEYOND THE STRING.
666B:          ;


666B:8E A0 66  IMPRINT STX  XSAV2      ; SAVE REGISTERS
666E:8C A1 66          STY  YSAV2      ;
6671:8D 9F 66          STA  ASAV2      ;

6674:68                PLA             ; GET POINTER LOW AND SAVE
6675:85 EB             STA  STRP2      ;
6677:68                PLA             ; GET POINTER HIGH AND SAVE
6678:85 EC             STA  STRP2+1    ;

667A:A0 00             LDY  #$00       ; NO INDEXING
667C:E6 EB     NXTCHR2 INC  STRP2      ; GET NEXT HIGH ADDRESS
667E:D0 02             BNE  NOC2       ; SKIP IF NO CARRY
6680:E6 EC             INC  STRP2+1    ; INCREMENT HIGH ADDRESS
6682:B1 EB     NOC2    LDA  (STRP2),Y  ; GET CHARACTER
6684:F0 09             BEQ  END2       ; IF ZERO MARKER
6686:20 9E 66          JSR  HOOK2      ; FOR SPECIAL EFFECTS
6689:20 ED FD          JSR  COUT       ; PRINT CHARACTER
668C:18                CLC             ; BRANCH ALWAYS
668D:90 ED             BCC  NXTCHR2    ;

668F:A5 EC     END2    LDA  STRP2+1    ; RESTORE PC LOW
6691:48                PHA             ;
6692:A5 EB             LDA  STRP2      ; RESTORE PC HIGH
6694:48                PHA             ;
6695:AE A0 66          LDX  XSAV2      ;
6698:AC A1 66          LDY  YSAV2      ; RESTORE REGISTERS
669B:AD 9F 66          LDA  ASAV2      ;
669E:60        HOOK2   RTS             ; AND EXIT



669F:          ;             *** STASH ***


669F:00        ASAV2   DFB  $00        ; ACCUMULATOR SAVE
66A0:00        XSAV2   DFB  $00        ; X-REGISTER SAVE
66A1:00        YSAV2   DFB  $00        ; Y-REGISTER SAVE

*** SUCCESSFUL ASSEMBLY: NO ERRORS
```

3

MONITOR TIME DELAY

how to use a monitor subroutine for sounds, animation, and other timing

If everyone is always worried about getting their programs to run fast enough, why on earth would you ever *purposely* want to stall for time?

Because, of course, some of the most useful and most interesting Apple uses center on carefully controlled sequences of time delays. The most obvious applications are in sound and music, where you wait for a while, and then change the position of a speaker cone. How long you wait sets the *pitch* of the tone, while the number of times you change the cone sets the *duration* of the note. Much more on this in the next two ripoff modules.

Another place where you purposely want to delay precise amounts of time involves baud-rate generation. Most often, though, these repeated time delays are done outside your CPU with a special serial transmitter chip. But other times, your CPU can be asked to generate a special frequency or a timing waveform that involves carefully controlled delays.

Producing the 40-kHz ultrasonic control signal for a BSR remote power controller is one use. Here a few bytes of software can replace bunches of specialized and unneeded hardware. Many industrial uses of Apples involve function and signal generators of one sort or another.

The real biggie of the time delay world centers on animation. To animate something, you put a pattern on the screen, wait a while, and then replace or modify that pattern into something different. Done just right, the changing patterns will give you the illusion of continuous motion. One very new use of Apple timing lets you carefully lock your animation to your video displays. This offers you everything from flawless and glitchless animation to mixing and matching of text, HIRES, and LORES together all at once.

Finally, there are the long term uses of time delay. Things that control appliances, turn on sprinklers, or that keep hourly, daily, weekly, or even monthly tabs on whatever it is that needs its tabs kept.

As with any programming technique, there are several different popular ways you can go about stalling for time. Which one you use depends on what you are trying to accomplish and how much else has to happen while the time delay is taking place.

The fundamental unit of Apple time delay is called a *clock cycle*. One clock cycle is roughly one microsecond, so you will need around one million of these for a one second delay. The Apple clock cycles are crystal controlled, so they are themselves accurate to at least one part in a million.

But there is one possible source of inaccuracy that will get to you if you aren't careful. Apple clock cycles are *not* precisely one microsecond long . . .

An Apple clock cycle is ROUGHLY 1 microsecond long.

An Apple clock cycle takes EXACTLY 0.978 microseconds or 978 nanoseconds.

A microsecond takes up EXACTLY 1.023 Apple clock cycles.

Just to confuse you further, these times are *average* values. Each 65th clock cycle is one-seventh longer than all the rest. This is done to uniquely solve a sticky timing glitch. The result is a tiny, and usually negligible, jitter in outside-world timing applications.

For most everyday needs, you simply say a cycle is a microsecond, and live with the two percent error you get. But, if you need an exact number of Apple clock cycles, or an exactly specified time delay, you have to "fine tune" your thinking to get precisely what you need.

As examples, locking to an Apple field takes a precise delay of 17030 Apple clock cycles, no more and no less. The time does not matter here; the cycles are everything. If you must have precisely one second of delay, you should use 1,022,727 clock cycles and not an even million. But never make things bunches more precise than you really need, since extra accuracy is often a pointless waste of time and effort.

I guess I really get into time delay techniques whole hog, since some of the most mind-blowing and most challenging Apple uses involve carefully controlled time delays where an exact result has to be gotten in an exact number of cycles. This, of course, is what most of the cheap video stuff was all about, (Sams 21524 and 21723) and is an ongoing challenge in the Enhance series (Sams 21822, etc.).

Sometimes you will only want to delay for a few clock cycles. Other times you might need great heaping bunches of cycles. So, you have a choice of time delay methods. Here, going from short to long, are some possible . . .

WAYS TO STALL FOR TIME

- cycle burner uppers
- simple loop
- monitor delay
- triple monitor delay
- combined use
- offloading

Burning up clock cycles is one good way for short time delays of a few microseconds. What you do is throw in some Apple CPU commands that don't really do anything but burn up clock cycles. These might be used to equalize two paths through time critical code, to provide video positioning, or be used anywhere else you need only a few cycles of correction.

Here are some standard . . .

CYCLE BURNER UPPERS

2 cycles . . .	NOP
3 cycles . . .	BCC taken or JMP
4 cycles . . .	NOP and NOP
5 cycles . . .	NOP and BCC taken
6 cycles . . .	NOP and NOP and NOP
7 cycles . . .	PHA and PLA

The object of the game is to use as few code bytes as possible for your delay and to not hurt anything else in the way of flags or working registers. You can find the "efficiency" of a 6502 instruction by dividing the number of cycles delayed by the number of bytes needed. NOPs are often your safest bet since they do the least damage.

Doing one single cycle of delay gets tricky. While many of the "illegal" commands in the 65C02 default to single cycle NOPs, there is no obvious way to do a single cycle delay with the older and stock 6502's. The way I usually handle a single delay cycle is to set up the *difference* between two paths that have even and odd total clock cycles. For instance, if your carry flag is set, a BCC takes up two cycles and a BCS takes up three.

If you do use branches for exact time delays, watch your page boundary crossings! A mysterious "extra" clock cycle or two will sometimes result if your code crosses a page when you didn't expect it to.

Once you get good at it, you should try to build your time delays into code commands, so that your code does other good stuff at the same time it is providing your time delay.

Needless to say, cycle burner uppers get old for more than a few clock cycles worth of delay. There are obviously better ways to stall for a second than by using 511,350 NOPs in a row.

What usually happens for longer delays is that you try to take up *most* of the delay with some efficient code, and then, if you have to, "equalize" with cycle burner uppers to hit any magic values you need.

The next larger arrow in our delay quiver is the simple loop. Like so . . .

```
                LDX #$06
LOOP            DEX
                BNE LOOP
```

What you have done is filled a loop with a value and then counted it down. Go through the math, and you will find you get a total of 5N+1 clock cycles. N here is the *hex* value you initially load the loop with. So this dude is good for 6, 11, 16, 21, 26, . . . clock cycles.

Note that a loop value of zero will go all the way around, rather than falling through, for a total of 1281 clock cycles. In terms of audio frequencies, this equals a square wave's half-period of just under 400 Hz. The reason the zero is missed is that it immediately is decremented to $FF and thus gets "caught" by the taken BNE branch. Zero is thus the *maximum* possible loop time.

For longer delays, you can put extra cycle burner uppers inside the loop, or else go to a loop within a loop. As examples, a NOP inside your loop changes the formula to 7N+1 cycles, while two simple loops inside each other will get you over a tenth of a second of delay.

But there is a much better way for medium-length delays. There is a super elegant and super versatile time delay built into the Apple monitor that is most useful for longer time delays. To use this routine, all you do is put a magic value into the accumulator and call the routine. Like this . . .

USING THE MONITOR TIME DELAY

1. Put a magic value in A.
2. Do a JSR to $FCA8.

And that's all there is to it. Go through the code on this, and you'll find it to be disgustingly elegant. All that gets used is the accumulator and two "borrowed" stack locations. Nothing else is tied up or used at all.

Part of the elegance involves the timing range you get. You can go anywhere from 13 clock cycles, on up to a sixth of a second, starting with only a single 8-bit magic value. Very conveniently, the available 256 time-delay values are spread out in a somewhat "log" fashion, so you get "tight" spacing on small delays and "wide" spacing on long delays.

The only reason this routine is not used as much as it should be is that the formula for the "magic" delay value is scary. Spooky even. And so misunderstood that even Apple has misprinted its formula in several different places.

The magic formula, expressed in *clock cycles* is . . .

```
MONITOR DELAY CYCLES  =  13 + 13.5*A  + 2.5*A*A
```

If you want the time delay in microseconds, just multiply the above result by 0.978.

Apple failed to do this on page 63 of the *Apple II Reference Manual* and on page 223 of the *Apple IIe Reference Manual.* To correct your manual, cross out "microseconds" and write in "clock cycles!"

For milliseconds, divide the scaled result by 1000, and for seconds of delay, divide by a million. As usual, don't forget to convert your decimal values into hex before assembling them, or your delay will end up wrong just about every time.

Since that formula is so ugly and nasty that it might even scare an eighth grader, we'll just spell it all out for you in longhand . . .

TIME DELAY VALUES FOR THE MONITOR WAIT SUBROUTINE

HEX A	DECIMAL A	CYCLES	MICROSECONDS	MILLISECONDS
$00	0	13	12	.012
$01	1	29	28	.028
$02	2	50	48	.048
$03	3	76	74	.074
$04	4	107	104	.104
$05	5	143	139	.139
$06	6	184	179	.179
$07	7	230	224	.224
$08	8	281	274	.274
$09	9	337	329	.329
$0A	10	398	389	.389
$0B	11	464	453	.453
$0C	12	535	522	.522
$0D	13	611	597	.597
$0E	14	692	676	.676
$0F	15	778	760	.76
$10	16	869	849	.849
$11	17	965	943	.943
$12	18	1066	1042	1.042
$13	19	1172	1145	1.145
$14	20	1283	1254	1.254
$15	21	1399	1367	1.367
$16	22	1520	1485	1.485
$17	23	1646	1608	1.608
$18	24	1777	1737	1.737
$19	25	1913	1869	1.869
$1A	26	2054	2007	2.007
$1B	27	2200	2150	2.15
$1C	28	2351	2298	2.298
$1D	29	2507	2450	2.45
$1E	30	2668	2608	2.608
$1F	31	2834	2770	2.77

TIME DELAY TABLE, CONTINUED

HEX A	DECIMAL A	CYCLES	MICROSECONDS	MILLISECONDS
$20	32	3005	2937	2.937
$21	33	3181	3109	3.109
$22	34	3362	3286	3.286
$23	35	3548	3468	3.468
$24	36	3739	3654	3.654
$25	37	3935	3846	3.846
$26	38	4136	4043	4.043
$27	39	4342	4244	4.244
$28	40	4553	4450	4.45
$29	41	4769	4661	4.661
$2A	42	4990	4877	4.877
$2B	43	5216	5098	5.098
$2C	44	5447	5324	5.324
$2D	45	5683	5555	5.555
$2E	46	5924	5790	5.79
$2F	47	6170	6031	6.031
$30	48	6421	6276	6.276
$31	49	6677	6526	6.526
$32	50	6938	6782	6.782
$33	51	7204	7042	7.042
$34	52	7475	7306	7.306
$35	53	7751	7576	7.576
$36	54	8032	7851	7.851
$37	55	8318	8130	8.13
$38	56	8609	8415	8.415
$39	57	8905	8704	8.704
$3A	58	9206	8999	8.999
$3B	59	9512	9298	9.298
$3C	60	9823	9602	9.602
$3D	61	10139	9911	9.911
$3E	62	10460	10224	10.224
$3F	63	10786	10543	10.543
$40	64	11117	10867	10.867
$41	65	11453	11195	11.195
$42	66	11794	11528	11.528
$43	67	12140	11867	11.867
$44	68	12491	12210	12.21
$45	69	12847	12558	12.558
$46	70	13208	12911	12.911
$47	71	13574	13268	13.268
$48	72	13945	13631	13.631
$49	73	14321	13999	13.999
$4A	74	14702	14371	14.371
$4B	75	15088	14748	14.748
$4C	76	15479	15130	15.13
$4D	77	15875	15518	15.518
$4E	78	16276	15910	15.91
$4F	79	16682	16306	16.306

TIME DELAY TABLE, CONTINUED

HEX A	DECIMAL A	CYCLES	MICROSECONDS	MILLISECONDS
$50	80	17093	16708	16.708
$51	81	17509	17115	17.115
$52	82	17930	17526	17.526
$53	83	18356	17943	17.943
$54	84	18787	18364	18.364
$55	85	19223	18790	18.79
$56	86	19664	19221	19.221
$57	87	20110	19657	19.657
$58	88	20561	20098	20.098
$59	89	21017	20544	20.544
$5A	90	21478	20995	20.995
$5B	91	21944	21450	21.45
$5C	92	22415	21911	21.911
$5D	93	22891	22376	22.376
$5E	94	23372	22846	22.846
$5F	95	23858	23321	23.321
$60	96	24349	23801	23.801
$61	97	24845	24286	24.286
$62	98	25346	24776	24.776
$63	99	25852	25270	25.27
$64	100	26363	25770	25.77
$65	101	26879	26274	26.274
$66	102	27400	26783	26.783
$67	103	27926	27298	27.298
$68	104	28457	27817	27.817
$69	105	28993	28341	28.341
$6A	106	29534	28869	28.869
$6B	107	30080	29403	29.403
$6C	108	30631	29942	29.942
$6D	109	31187	30485	30.485
$6E	110	31748	31034	31.034
$6F	111	32314	31587	31.587
$70	112	32885	32145	32.145
$71	113	33461	32708	32.708
$72	114	34042	33276	33.276
$73	115	34628	33849	33.849
$74	116	35219	34427	34.427
$75	117	35815	35009	35.009
$76	118	36416	35597	35.597
$77	119	37022	36189	36.189
$78	120	37633	36786	36.786
$79	121	38249	37389	37.389
$7A	122	38870	37996	37.996
$7B	123	39496	38608	38.608
$7C	124	40127	39224	39.224
$7D	125	40763	39846	39.846
$7E	126	41404	40473	40.473
$7F	127	42050	41104	41.104

TIME DELAY TABLE, CONTINUED

HEX A	DECIMAL A	CYCLES	MICROSECONDS	MILLISECONDS
$80	128	42701	41740	41.74
$81	129	43357	42382	42.382
$82	130	44018	43028	43.028
$83	131	44684	43679	43.679
$84	132	45355	44335	44.335
$85	133	46031	44996	44.996
$86	134	46712	45661	45.661
$87	135	47398	46332	46.332
$88	136	48089	47007	47.007
$89	137	48785	47688	47.688
$8A	138	49486	48373	48.373
$8B	139	50192	49063	49.063
$8C	140	50903	49758	49.758
$8D	141	51619	50458	50.458
$8E	142	52340	51163	51.163
$8F	143	53066	51872	51.872
$90	144	53797	52587	52.587
$91	145	54533	53306	53.306
$92	146	55274	54031	54.031
$93	147	56020	54760	54.76
$94	148	56771	55494	55.494
$95	149	57527	56233	56.233
$96	150	58288	56977	56.977
$97	151	59054	57726	57.726
$98	152	59825	58479	58.479
$99	153	60601	59238	59.238
$9A	154	61382	60001	60.001
$9B	155	62168	60770	60.77
$9C	156	62959	61543	61.543
$9D	157	63755	62321	62.321
$9E	158	64556	63104	63.104
$9F	159	65362	63892	63.892
$A0	160	66173	64685	64.685
$A1	161	66989	65482	65.482
$A2	162	67810	66285	66.285
$A3	163	68636	67092	67.092
$A4	164	69467	67905	67.905
$A5	165	70303	68722	68.722
$A6	166	71144	69544	69.544
$A7	167	71990	70371	70.371
$A8	168	72841	71203	71.203
$A9	169	73697	72040	72.04
$AA	170	74558	72881	72.881
$AB	171	75424	73728	73.728
$AC	172	76295	74579	74.579
$AD	173	77171	75435	75.435
$AE	174	78052	76297	76.297
$AF	175	78938	77163	77.163

TIME DELAY TABLE, CONTINUED

HEX A	DECIMAL A	CYCLES	MICROSECONDS	MILLISECONDS
$B0	176	79829	78034	78.034
$B1	177	80725	78910	78.91
$B2	178	81626	79790	79.79
$B3	179	82532	80676	80.676
$B4	180	83443	81566	81.566
$B5	181	84359	82462	82.462
$B6	182	85280	83362	83.362
$B7	183	86206	84267	84.267
$B8	184	87137	85177	85.177
$B9	185	88073	86092	86.092
$BA	186	89014	87012	87.012
$BB	187	89960	87937	87.937
$BC	188	90911	88867	88.867
$BD	189	91867	89801	89.801
$BE	190	92828	90740	90.74
$BF	191	93794	91685	91.685
$C0	192	94765	92634	92.634
$C1	193	95741	93588	93.588
$C2	194	96722	94547	94.547
$C3	195	97708	95511	95.511
$C4	196	98699	96479	96.479
$C5	197	99695	97453	97.453
$C6	198	100696	98432	98.432
$C7	199	101702	99415	99.415
$C8	200	102713	100403	100.403
$C9	201	103729	101396	101.396
$CA	202	104750	102394	102.394
$CB	203	105776	103397	103.397
$CC	204	106807	104405	104.405
$CD	205	107843	105418	105.418
$CE	206	108884	106435	106.435
$CF	207	109930	107458	107.458
$D0	208	110981	108485	108.485
$D1	209	112037	109518	109.518
$D2	210	113098	110555	110.555
$D3	211	114164	111597	111.597
$D4	212	115235	112644	112.644
$D5	213	116311	113695	113.695
$D6	214	117392	114752	114.752
$D7	215	118478	115814	115.814
$D8	216	119569	116880	116.88
$D9	217	120665	117952	117.952
$DA	218	121766	119028	119.028
$DB	219	122872	120109	120.109
$DC	220	123983	121195	121.195
$DD	221	125099	122286	122.286
$DE	222	126220	123382	123.382
$DF	223	127346	124482	124.482

TIME DELAY TABLE, CONTINUED

HEX A	DECIMAL A	CYCLES	MICROSECONDS	MILLISECONDS
$E0	224	128477	125588	125.588
$E1	225	129613	126698	126.698
$E2	226	130754	127814	127.814
$E3	227	131900	128934	128.934
$E4	228	133051	130059	130.059
$E5	229	134207	131189	131.189
$E6	230	135368	132324	132.324
$E7	231	136534	133464	133.464
$E8	232	137705	134608	134.608
$E9	233	138881	135758	135.758
$EA	234	140062	136913	136.913
$EB	235	141248	138072	138.072
$EC	236	142439	139236	139.236
$ED	237	143635	140405	140.405
$EE	238	144836	141579	141.579
$EF	239	146042	142758	142.758
$F0	240	147253	143942	143.942
$F1	241	148469	145130	145.13
$F2	242	149690	146324	146.324
$F3	243	150916	147522	147.522
$F4	244	152147	148726	148.726
$F5	245	153383	149934	149.934
$F6	246	154624	151147	151.147
$F7	247	155870	152365	152.365
$F8	248	157121	153588	153.588
$F9	249	158377	154816	154.816
$FA	250	159638	156048	156.048
$FB	251	160904	157286	157.286
$FC	252	162175	158528	158.528
$FD	253	163451	159776	159.776
$FE	254	164732	161028	161.028
$FF	255	166018	162285	162.285

A copy of this listing appears on the companion diskette as a bonus program. Make as many copies as you like in any format you care to.

We'll find out just how to use the monitor delay subroutine shortly. Note that you do not get *every* value in the range you need. What you do is take the nearest value and then either live with it or else "pad" it with cycle burner uppers.

Let's quickly round out our survey of ways to stall for time. If you use the monitor delay three times in a row with just the right *different* "magic" values, you can hit practically any exact value over a one to three hundred millisecond range. This was needed and used extensively in *Enhancing Your Apple II* (Sams 21822).

As another bonus program on the support diskette for this book, we'll throw in an automatic magic number finder that quickly solves the triple delay problem for you. The task is not trivial. More details on this support diskette are found inside the back cover.

On longer delays, it is always best to try and do other things while you are stalling for time. For instance, you can increment a random number pair while you are waiting for someone to press a key. Or you can use your animated graphics plotting time as part of the time delay for a sound. Always suspect long times spent "wheel spinning," and see if you can't replace stalling code with some useful yet time consuming task instead . . .

Avoid "wheel spinning" for wheel spinning's sake.

ALWAYS try and make your time delay code handle other useful tasks.

The "best" way to stall for time is to have something other than the CPU do the delaying for you. This frees up your Apple to go on to do other useful things. For instance, you can send a single and fast "transmit" command to a serial card whose separate UART takes its good old time outputting a serial code. Or, send your music commands to a music chip. Or your timer commands to a timer chip. Or use a real time clock chip to interrupt your Apple for those things that take really long time delays, such as control of a sprinkler system.

Unfortunately, all of these "offloaders" take special hardware and add to your system cost. They also limit who you can sell your product to. Sometimes it is best to do your initial timing with the CPU and then later offload cumbersome timing once your product is better defined.

Using the Monitor Delay

Let's find out how to use the monitor delay for some exciting and noisy animation. So stunning, in fact, that it might earn a fifth grader a B− if his teacher was feeling generous. While we are at it, we will pick up some fundamentals of LORES plotting using the existing monitor LORES subs.

Many people look down on LORES, but a thorough understanding of LORES graphics is almost essential if you are ever going to handle HIRES. The IIe now offers double LORES graphics of 24 × 80 color blocks, which considerably eases the "chunkiness" of the display.

LORES animation and repeated mapping can be done much faster and with far fewer bytes than can be done in HIRES. And, thanks to the exact field sync of the Enhancing series, you can easily mix and match text, LORES, and HIRES together anyplace you want on the screen all at the same time.

Our main program is called DEMO3. DEMO3 consists of three subroutines, just as any "high level" code should be made up entirely of subroutine calls. The first subroutine clears the screen and draws an empty bucket on the screen. The second subroutine fills the bucket at a one layer per second rate. The third subroutine causes an explosion when the bucket is completely filled. Calling the fire department or pressing any key ends the explosion.

We will let you do your own flowchart on this, since nothing sneaky is involved.

The first subroutine is called DRAWCUP. This one initializes the LORES screen and clears it using the existing SETGR and CLRSCR monitor subs. You then set the bucket color to green using the SETCOL subroutine, and then draw your bucket.

Bucket drawing is done using the HLINE and VLINE monitor subroutines. You enter HLINE with the vertical position in the accumulator, the left end line position in the Y register, and the right end line position in page zero location $2C.

Alike but different somehow, you enter VLINE with the horizontal position in the Y register, the top-most line position in the accumulator, and the bottom-most line position in page zero location $2D.

Note how the use of labels HEND for $2C and VBOT for $2D eases remembering these values.

The FILLCUP subroutine fills the cup one level at a time, spending one second per level. Several sub-subs are involved. The TENTHS subroutine uses the monitor delay to produce one-tenth of a second delay. In this demo, we won't worry about exact timing values, since they are not at all critical.

Since we cannot do a one-second delay directly with the monitor sub, we instead use our own SECONDS subroutine, which calls the TENTHS subroutine ten times in a row to get a one-second delay.

To round out our time delays, there is also a TENMSEC subroutine that generates a 10-millisecond delay, useful to produce a sound effect as part of the BRACK subroutine. More details on sound effects appear in the next two ripoff modules.

The "explosion" is done by rapidly changing the screen modes while whapping the speaker. It sounds and looks awful.

MIND BENDERS

—Why does the liquid stay inside the cup, rather than overwriting the existing cup sides?

—What are the *exact* time delays in use, including all sub timing and all overhead code?

—Improve the animation and the display so it would earn a seventh grader an A−.

—Only certain cup and liquid colors are compatible on an average color set. Why? Which combinations look best in both color and black and white?

—Redo this demo in HIRES. Do an on-screen splash. Then include a *real* squirt gun in your demo.

PROGRAM RM-3
MONITOR TIME DELAY

```
----- NEXT OBJECT FILE NAME IS TIME DELAY
6700:                 ORG  $6700     ; PUT MODULE #3 AT $6700


6700:            ;   ****************************************
6700:            ;   *                                      *
6700:            ;   *            -< TIME DELAY >-          *
6700:            ;   *                                      *
6700:            ;   *          (USING MONITOR WAIT)        *
6700:            ;   *                                      *
6700:            ;   *      VERSION 1.0   ($6700-$67AC)     *
6700:            ;   *                                      *
6700:            ;   *               11-24-82               *
6700:            ;   *......................................*
6700:            ;   *                                      *
6700:            ;   *          COPYRIGHT C 1982 BY         *
6700:            ;   *                                      *
6700:            ;   *     DON LANCASTER AND SYNERGETICS    *
6700:            ;   *     BOX 1300, THATCHER AZ., 85552    *
6700:            ;   *                                      *
6700:            ;   *     ALL COMMERCIAL RIGHTS RESERVED   *
6700:            ;   *                                      *
6700:            ;   ****************************************


6700:            ;            *** WHAT IT DOES ***

6700:            ;   THIS PROGRAM SHOWS HOW TO USE THE MONITOR WAIT
6700:            ;   SUBROUTINE FOR TIME DELAYS OF 0.01, 0.1, 1.0,
6700:            ;   AND 10.0 SECONDS.
6700:            ;
6700:            ;
6700:            ;


6700:            ;            *** HOW TO USE IT ***

6700:            ;   TO USE, RUN THE DEMO BY $6700G FROM MACHINE LANGUAGE
6700:            ;   OR CALL 26368 FROM APPLESOFT.
6700:            ;
6700:            ;   THEN ADAPT THE METHOD AND RESULTS TO YOUR OWN
6700:            ;   NEEDS.
6700:            ;
```

PROGRAM RM-3, CONT'D . . .

```
6700:          ;           *** GOTCHAS ***

6700:          ;   THE ACCUMULATOR IS DESTROYED BY THE WAIT SUBROUTINE.
6700:          ;
6700:          ;   MACHINE TIME AND PEOPLE TIME DIFFER! ONE CLOCK CYCLE
6700:          ;   EQUALS 0.976 MICROSECONDS, AND NOT 1.000 MICROSECONDS!
6700:          ;
6700:          ;   THIS SLIGHT DIFFERENCE CAN SOMETIMES BE SIGNIFICANT.


6700:          ;          *** ENHANCEMENTS ***

6700:          ;   DEMO3 ALSO SHOWS YOU SEVERAL TRICKS INVOLVED WHEN
6700:          ;   YOU USE THE LORES SCREEN.
6700:          ;
6700:          ;
6700:          ;
6700:          ;


6700:          ;          *** RANDOM COMMENTS ***

6700:          ;   IF YOU NEED AN EXACT NUMBER OF MACHINE CYCLES THAT
6700:          ;   CANNOT BE HIT DIRECTLY WITH WAIT, TRY USING WAIT TWO
6700:          ;   OR THREE TIMES USING DIFFERENT A VALUES.
6700:          ;
6700:          ;
6700:          ;
```

PROGRAM RM-3, CONT'D . . .

```
6700:            72 ;           *** HOOKS ***

F832:            74 CLRSCR  EQU  $F832     ; CLEAR FULL LORES SCREEN
002C:            75 HEND    EQU  $2C       ; RIGHT END OF LORES H LINE
C057:            76 HIRES   EQU  $C057     ; HIRES SOFT SWITCH
F819:            77 HLINE   EQU  $F819     ; HORIZ LORES LINE
FB2F:            78 INIT    EQU  $FB2F     ; INITIALIZE TEXT SCREEN
C000:            79 IOADR   EQU  $C000     ; KEYBOARD INPUT
C010:            80 KBDSTR  EQU  $C010     ; KEYSTROBE RESET
C056:            81 LORES   EQU  $C056     ; LORES SOFT SWITCH
C053:            82 LOWSCR  EQU  $C053     ; PAGE ONE SOFT SWITCH
C052:            83 MIXCLR  EQU  $C052     ; FULL GRAPHICS SCREEN
F864:            84 SETCOL  EQU  $F864     ; SET LORES COLOR
FB40:            85 SETGR   EQU  $FB40     ; SET UP GRAPHICS SCREEN
C030:            86 SPKR    EQU  $C030     ; SPEAKER CLICK OUTPUT
C050:            87 TXTCLR  EQU  $C050     ; GRAPHICS ON SOFT SWITCH
C051:            88 TXTSET  EQU  $C051     ; TEXT ON SOFT SWITCH
002D:            89 VBOT    EQU  $2D       ; BOTTOM OF LORES V LINE
F828:            90 VLINE   EQU  $F828     ; VERTICAL LORES LINE
FCA8:            91 WAIT    EQU  $FCA8     ; TIME DELAY SET BY ACCUMULATOR
```

PROGRAM RM-3, CONT'D . . .

```
6700:              ;            *** DEMO ***
6700:              ;
6700:              ;

6700:              ;       THE DEMO FILLS A LORES BUCKET
6700:              ;       EACH SECOND TILL OVERFLOW,
6700:              ;       TICKING OFF EACH TENTH OF A SECOND.
6700:              ;
6700:              ;
6700:              ;

6700:20 0A 67      DEMO3   JSR  DRAWCUP   ; DRAW LORES CUP
6703:20 3F 67              JSR  FILLCUP   ; FILL CUP
6706:20 5C 67              JSR  EXPLODE   ; THEN EXPLODE
6709:60                    RTS            ; AND EXIT



670A:              ;             *** DRAWCUP SUBROUTINE ***
670A:              ;
670A:              ;     THE DRAWCUP SUBROUTINE DRAWS A LORES CUP ON THE SCREEN.
670A:              ;
670A:              ;
670A:              ;

670A:20 40 FB      DRAWCUP JSR  SETGR     ; INIT LORES SCREEN
670D:2C 52 C0              BIT  MIXCLR    ; FULL SCREEN GRAPHICS
6710:20 32 F8              JSR  CLRSCR    ; CLEAR FULL LORES SCREEN
6713:A9 04                 LDA  #$04      ; USE GREEN BUCKET
6715:20 64 F8              JSR  SETCOL    ; AND SET COLOR
6718:A9 19                 LDA  #$19      ; DRAW BASE
671A:85 2C                 STA  HEND      ;
671C:A0 0C                 LDY  #$0C      ;
671E:A9 1E                 LDA  #$1E      ;
6720:20 19 F8              JSR  HLINE     ; AND PLOT IT
6723:A9 1E                 LDA  #$1E      ; DRAW SIDES
6725:85 2D                 STA  VBOT      ;
6727:A0 0D                 LDY  #$0D      ;
6729:A9 14                 LDA  #$14      ;
672B:20 28 F8              JSR  VLINE     ; AND DRAW LEFT SIDE
672E:A9 14                 LDA  #$14      ;
6730:A0 18                 LDY  #$18      ;
6732:20 28 F8              JSR  VLINE     ; AND DRAW RIGHT SIDE
6735:A9 06                 LDA  #$06      ; SET COLOR FOR FILL
6737:20 64 F8              JSR  SETCOL    ;
673A:C6 2C                 DEC  HEND      ; FILL INSIDE RIGHT
673C:C6 2C                 DEC  HEND
673E:60                    RTS            ; AND RETURN
```

PROGRAM RM-3, CONT'D . . .

```
673F:           ;     *** FILLCUP SUBROUTINE ***
673F:           ;
673F:           ;   THIS SUBROUTINE FILLS THE CUP AT
673F:           ;   A ONE SECOND PER LEVEL RATE.
673F:           ;
673F:           ;

673F:A9 0A      FILLCUP LDA  #$0A       ; FOR TEN TRIPS
6741:8D AC 67           STA  CUPHI      ;  SAVE INDEX
6744:20 53 67   AGAIN3  JSR  SECONDS    ; DELAY VIA SECONDS SUB
6747:20 A0 67           JSR  POUR       ; ADD TO LEVEL
674A:20 86 67           JSR  BRACK3     ; MAKE NOISE
674D:CE AC 67           DEC  CUPHI      ; NEXT CUP LEVEL
6750:D0 F2              BNE  AGAIN3
6752:60                 RTS             ; AND EXIT



6753:           ;     *** SECONDS SUBROUTINE ***

6753:           ;
6753:           ;
6753:           ;
6753:           ;

6753:A0 0A      SECONDS LDY  #$0A       ; FOR TEN TENTHS
6755:20 94 67   NEXT3   JSR  TENTHS     ; DELAY FOR A TENTH
6758:88                 DEY             ;
6759:D0 FA              BNE  NEXT3      ; REPEAT TILL DONE
675B:60                 RTS             ; THEN EXIT
```

PROGRAM RM-3, CONT'D . . .

```
675C:          ;     *** EXPLODE SUBROUTINE ***


675C:2C 57 C0  EXPLODE BIT  HIRES      ;
675F:20 9A 67          JSR  TENMSEC    ; DELAY FOR TEN  MILLISECONDS
6762:2C 56 C0          BIT  LORES
6765:20 9A 67          JSR  TENMSEC    ; AND DELAY AGAIN
6768:2C 51 C0          BIT  TXTSET     ;
676B:20 9A 67          JSR  TENMSEC    ;
676E:2C 30 C0          BIT  SPKR       ;
6771:2C 50 C0          BIT  TXTCLR     ;
6774:20 9A 67          JSR  TENMSEC    ;
6777:2C 30 C0          BIT  SPKR       ; WHAP SPEAKER
677A:2C 00 C0          BIT  IOADR      ; CHECK FOR KEYPRESS
677D:10 DD             BPL  EXPLODE    ;
677F:2C 10 C0          BIT  KBDSTR     ; RESET KEYBOARD
6782:20 2F FB          JSR  INIT       ; BACK TO TEXT SCREEN
6785:60                RTS             ; POP STACK AND RETURN



6786:          ;        *** BRACK SUBROUTINE ***


6786:A0 06     BRACK3  LDY  #$06       ; SECONDS TONE
6788:A9 0C     NOTE3   LDA  #$0C       ;
678A:20 A8 FC          JSR  WAIT       ;
678D:2C 30 C0          BIT  SPKR       ;
6790:88                DEY             ;
6791:D0 F5             BNE  NOTE3      ;
```

PROGRAM RM-3, CONT'D . . .

```
6793:60                 RTS           ;

6794:              ;      *** TENTHS SUBROUTINE ***
6794:              ;
6794:              ;   THIS SUB USES WAIT TO DELAY ONE TENTH
6794:              ;   OF A SECOND.

6794:A9 C7         TENTHS  LDA  #$C7       ; FOR 99.415 MILLISECONDS
6796:20 A8 FC              JSR  WAIT       ; DELAY VIA WAIT SUB
6799:60                    RTS             ; AND THEN RETURN



679A:              ;      *** TEN MILLISECONDS SUB ***
679A:              ;
679A:              ;   THIS SUB USES WAIT TO DELAY TEN MILLISECONDS.
679A:              ;

679A:A9 3D         TENMSEC LDA  #$3D       ; FOR 99.415 MILLISECONDS
679C:20 A8 FC              JSR  WAIT       ; DELAY VIA WAIT SUB
679F:60                    RTS             ; AND THEN RETURN



67A0:              ;     *** POUR SUBROUTINE ***
67A0:              ;

67A0:18            POUR    CLC             ; FILL CUP WITH LIQUID
67A1:A9 13                 LDA  #$13       ; TOP OF CUP LEVEL
67A3:6D AC 67              ADC  CUPHI      ;  MINUS HEIGHT ALREADY
67A6:A0 0E                 LDY  #$0E       ; LEFT SIDE SET
67A8:20 19 F8              JSR  HLINE      ; DRAW LEVEL
67AB:60                    RTS             ; AND EXIT


67AC:              ;     *** STASH ***

67AC:0A            CUPHI   DFB  $0A        ; LEVEL IN CUP




*** SUCCESSFUL ASSEMBLY: NO ERRORS
```

4

OBNOXIOUS SOUNDS

an extremely versatile and compact wide-range sound-effects generator

First the bad news.

For a given amount of programming effort and add-on hardware, the Apple will always give you sound that is "thin" and animation that is "weak," when compared against an arcade video game. This happens inevitably because the Apple CPU has to take time out to generate its own sound and graphics, and because the color system is stuck with being more or less compatible with the NTSC ("Never The Same Color") broadcast television standard.

The good news, of course, is that for an extraordinary amount of creative programming effort, and for super creative use of extra hardware, you can use your Apple to knock the bytes out of *any* arcade video game or *any* other brand of personal computer. All it takes is lots of special effort that optimizes what you can do within the bounds of the actual limits of your Apple.

The next two ripoff modules show us two of the many different ways you can get sound into your programs. We will assume, for now, that you are going to use the built-in speaker of an unmodified Apple.

The "noisemaking" hardware of your Apple seems rather limiting at first glance. You have one small and tinny-sounding speaker. All the support hardware lets you do is shove the speaker cone all the way in, or else pull it all the way out. You do this by "whapping" address location $C030 *once* each time you want to *change* the cone's position.

Technically, address $C030 is decoded and used to change the state of a binary divider, or flip-flop. The flip-flop is coupled to a special Darlington driver transistor. One whap pushes the cone in. The next pulls it out . . .

A BIT $C030 is the standard way of moving the Apple speaker's cone from the extreme position it is in to the other extreme position.

To get some useful sounds out of the Apple speaker, you decide how often you want to shove the cone back and forth, and carefully pick the time delay needed between shovings. For instance, if you keep a constant time between shovings, you will set the pitch of an audio square wave. The duration of the tone is decided by how long you continue the shoving process.

Believe it or not, you can easily get more than one note at once, have variable volume, do bell-like tones, handle speech, and do much, much more if you are a sneaky enough programmer. And your sound can be further "thickened" considerably just by adding a larger speaker to your Apple.

Before getting fancy, though, let's get two gotchas out of the road . . .

$C030 GOTCHAS

two whaps immediately following each other give you no sound . . .

USE BIT $C030, **NOT** STA $C030.

one isolated whap may not sound . . .

USE AT LEAST 3 WHAPS PER CLICK.

Due to a quirk in the Apple's timing, any time you write to a memory location, you address that location twice. The two addressings are separated by one microsecond. If you try to do a STA $C030, you end up shoving the speaker in and then pulling it back out again an impossibly brief time later. The cone barely moves in so short a time, and, surprise, surprise, you get no sound.

So, always BIT test your speaker location. Do not write to it, unless you want no sound.

The second quirk comes about because of an attempt to save Apple system power. There is a coupling capacitor in the path between the flip-flop and the speaker. This capacitor discharges on inactivity. Which means that the speaker cone is never held "in" for long periods of time. The dropout time is long compared to most tones, so you normally won't notice it.

There are two places where you might pick up the side effects of this power-down capacitor, and where it may cause you trouble. If you try to "click" the speaker just once, there's only a 50-50 chance you will get any sound at all. So, it takes *two* repeated commands, delayed by some audio value, to guarantee an isolated click. In real life, three or four repeated speaker motions are the minimum you will want to use, since some of the clicks will sound "leaner" than others.

The other place this gotcha appears happens when you send very low-pitched notes, or a very low-pitch "sweep" to your speaker. At some point, the frequency will jump up by an octave. This frequency doubling happens when the capacitor picks up enough charge to allow cone clicking in both directions.

Watch these two details if you ever get no sound or uneven sound out of your code.

As in all other Apple programming techniques, there are lots of different ways to get sound, and each of these ways will have a certain range of effects over which they are useful. Let's survey some . . .

WAYS TO GENERATE APPLE SOUND

clickety clack
calculated routine
red book tones

table method
duty cycling
offloading

With the *clickety clack* method, you simply move the speaker cone back and forth a few times, using a loop or some other obvious code. See the BRACK subroutine of the previous ripoff module for an example. The time between whappings sets the pitch while the total number of whappings sets the duration of your sound. If the pitch is constant, you get a "pure" tone. If the pitch changes, you get a "sweep." If both halves of each cycle are the same time duration, you get a "woodwind" style tone. If one half of each cycle is much longer than the other, you get a "string" style voicing.

One important exception to the clickety clackers. Do not *ever* use the standard "[G]" or "JSR $FF3A" beep. This tone is too grating to ever use in any reasonable program . . .

NEVER use the "standard" Apple beep anywhere in any of your programs!

ALWAYS kick sand in the face of anyone who does.

In the *calculated routine* method, you generate some code that decides when and where all the zero crossings are needed for a certain sound effect. This method is often used for sirens and sweeps, tonal scales, frog croaks, phasors, and other short or weird "one-shot" sounds.

The good thing about the calculated routine method is that you can get some real serendipity going, and end up with some totally wild sounds that you wouldn't ever have thought possible otherwise. The bad scene about many calculated routines is that this is "old" code done the "old" way that may end up long and cumbersome, rather than short and general.

The OBNOXIOUS SOUNDS subroutine of this ripoff module will shortly explore this technique.

The *red book tones* method is a way to make monophonic music that is useful for playing songs in tempered musical scales. This involves a pitch and duration generator, and some file access tricks. More on this in the next module.

The *table method* looks up each speaker motion as needed, out of a long table. You can produce *any* possible sound this way. Most Apple-based speech uses the table method, and virtually any sound of most any complexity can be handled with a general and versatile enough program.

There are some tricks to using the table method. Getting the table to sound like you really want it to can be very involved and may take a long time. Finding some suitable coding that lets you put lots of sound in a short table is also a real hassle. Long or multiple effects really burn up the bytes. The obvious brute force method of storing a one each time you want the speaker to move can be substantially improved by going to some sort of "run length" encoding.

The best way to study table method sound is to steal the German vocabulary file out of *Castle Wolfenstein.* To grab this table, just follow the "tearing" method of Enhancement 3 in the *Enhancing Your Apple II,* Volume I (Sams 21822).

Ah yes. Duty Cycling.

Pushing the limits. Doing the impossible. How on earth can you get more than one tone at a time out of a speaker driver that you can only push or pull? How can you do variable volume? Sinewaves and flute-like or bell-like tones?

Its really very simple. Suppose you *extremely rapidly* move the speaker cone in and out, at an ultrasonic rate. The *average* cone position depends on the *average* duty cycle. For a sinewave, just let the average cone position describe a sinewave at the frequency you want. For bell tones, let the average position slowly "decay" to its "middle" value. For more than one note at once, just let the average position equal the sum of all the notes taken together at once.

If you get into some hairy math involving *Fourier coefficients,* you can easily handle chords and other multitone effects, with or without duty cycling. The whole trick is to, on the average, put the speaker cone where it ought to be when it ought to be there.

Duty cycling techniques are described in various issues of *Apple Assembly Line.*

Offloading consists of using something other than the Apple's speaker to make the noise. Simply going to a larger speaker or into a hi-fi will help "thicken" the sound bunches, and you can get stereo effects by using the speaker hardware for one channel and the cassette output port for the other. You can separately get four more channels out of the annunciator outputs of your game paddle connector.

But the real benefits of offloading take place when you send simple commands to a custom noise generator or music generator chip. Besides producing much richer and more flexible sounds, you now offload the Apple's CPU so it is free to go on to other things. All the Apple has to do is quickly pass a few parameters on to the music chip, rather than stalling around for the entire time it takes to produce the entire tone or tone sequence.

Both *General Instruments* and *Texas Instruments* are heavily into music and sound-effect generation chips. These are often the key circuits used in the fancier plug-in synthesizer cards and systems as well.

Time now for more details on . . .

The Calculated Routine Method

The calculated routine method is best done for single and isolated sound effects.

A phasor blast, of course, is the architypical example of this sort of thing. We'll show you a few dozen bytes of code that do the standard and classical phasor blast for you. But, by changing only two values, those same bytes can do a surprising variety of effects that sound wildly different.

These include some very pleasant and highly "brassy" prompt tones, musical glissades, some "cartoon" style sound, a geiger-counter simulation, and a few assorted and highly useful pips, ticks, and whopidoops. There's even a special effect called the *time bomb,* that lasts for minutes, and has all sorts of impractical joke possibilities.

The object of *any* sound program is to produce some speaker whappings separated by some time delays. The time delays set the time between zero crossings of the sound that the speaker is to produce. Usually these delays will range from 10 microseconds to 10 milliseconds or so. Faster than this and you are into ultrasonics that you cannot hear and that the speaker cone cannot follow. Slower than this breaks the sound down into individual and possibly annoying clicks.

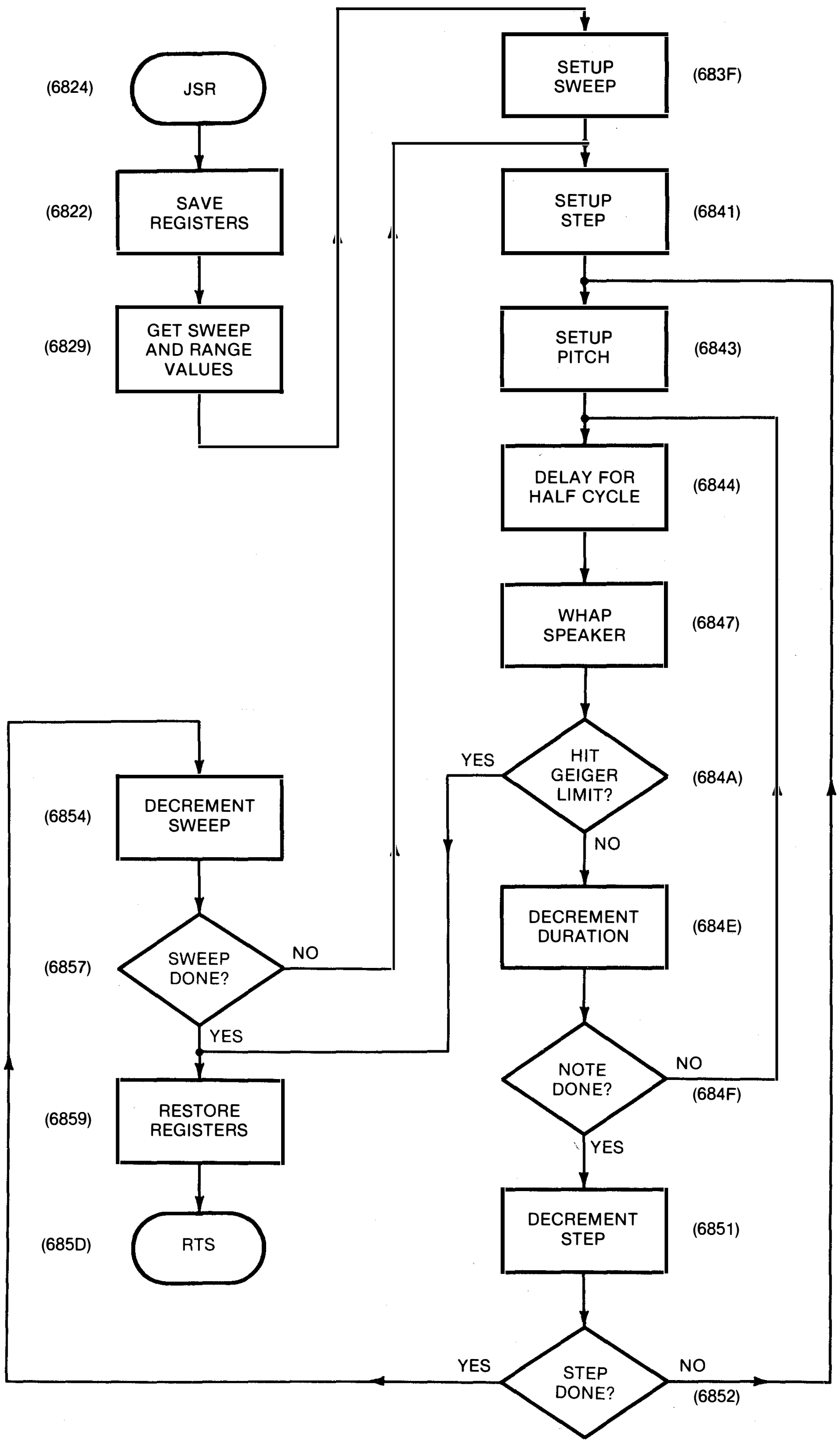
If all the time delays are the same value, you get a constant square-wave tone. The total number of time delays sets the duration of the tone, while each individual delay sets the pitch of each half-cycle of sound.

Things get interesting when you vary the time delays in a strange manner. For instance, if you make each successive time delay shorter, you get a siren or *sweep* effect that goes up in time.

The whole intent of the calculated routine method is to produce some interesting changes in the time delays that give you fat, thick, and interesting sound effects. The calculations of your routine should create a group of delay values that result in a useful sound.

Here's the flowchart for this module's calculated routine sound effects generator . . .

OBNOXIOUS SOUNDS FLOWCHART:



Actually, this is nothing but a very simple sweep generator with one or two added tricks. The only two parameters under your control are how *far* you sweep and the *total number of sweeps* you use. Now, don't go away, for you will be utterly amazed at how many *totally different* effects you can get this way. In theory, there are 65536 different effects possible. In practice, there's only two dozen or so that you will find genuinely useful and uniquely different.

Our first trick is to use the monitor delay subroutine. Remember that these delay values are "cramped together" at the short end, giving you a more or less log response. And this is just what you want for an audio sweep. A linear sweep sounds awful, since your ear is a log device that expects a few cycles change for low notes and lots of cycles change for high notes. So, the monitor delay sub automatically puts the low notes close together and the high notes far apart, just like you need.

Our second trick is to use the same value to set both the pitch and the length of each step in the sweep. This keeps things simple, yet still gives you many different sounds.

Our third trick is very sneaky. Five testing bytes are added to give you geiger counter or multiple click effects. If the sweep duration is *less* than $80, you get the complete sweep, all the way up in pitch. If the sweep duration is *greater* than $80, the sweep only goes to the $80 value and then quits.

The $80 value is extremely low in pitch. So low that you hear each cone movement as a distinct click. With the five byte code patch, values greater than $80 give you a burst of clicks. Values less than $80 give you the full sweep. So, you get two wildly and totally different classes of sound effects out of the same simple calculated routine.

We have used a sixteen-entry file to support the sound effects generator. If you only want one or two sounds, you can eliminate this file and direct poke the effects you are after. Each sound effect is specified with two values. The first decides the *number* of the sweeps produced, while the second decides *how long* each sweep is to be.

At any rate, you enter the subroutine with a number in the X register that equals the sound effect you are after. You then save all the other registers. Next you check to make sure the number is legal. If it is not, you replace it with sound effect zero. You might prefer some fancier error trapping here, but this is probably all you will really need.

Next, the sound effect number is converted into two sweep values by looking them up in the SEF effects file. The number of sweeps is grabbed first and put in an absolute location called TRPCNT4. This location will get counted down, once per each complete sweep. After this, the sweep duration is grabbed and "force fed" into the code at SWEEP4+1.

Uh, whoops. Play that one by again.

Tricks like this go by the name of *self-modifying* code. Which is legal and powerful if you know what you are doing. What you have done here is *changed* a LDY #00 command into a LDY #SWEEPS command. Note that the data value gets poked into the *second* byte of the op code! Put it anywhere else and you plow the program. Note also that any self-modifying code *must* be in RAM. EPROM need not apply.

Why?

Generally, it is safe to *pre-modify* your code like we have done here. In fact, this is a standard and powerful programming technique. Just be sure that you are changing ONLY the EXACT location you think you are. On the other hand, code that continuously changes itself on the fly is very dangerous. Deadly even. Yet still a specialized and most useful programming technique.

Some comment . . .

If you self-modify code, be sure to place what you are putting EXACTLY where you intend to put it!

One or two data values preplaced once before use is safe and standard.

Code that continuously changes itself is often dumb and deadly.

So much for a side trip on self-modifying code. At this point, we have a "number-of-sweeps" value in TRPCNT4, and the "length-of-the-sweep" value has been force fed into a command that loads the Y register.

Now to get sneaky. We need a third parameter. Namely, the duration value that sets the frequency for this part of our sweep. For simplicity, we just transfer Y to X, and let X set our duration and Y our pitch. For any given step of our sweep, we want all constant frequencies. Thus, we will keep Y constant while we count X down. This results in a sound that sweeps up in distinct note-like steps.

So far, so good. You transfer your pitch value to the accumulator and then use the monitor delay to stall for a half-cycle. Then, you whap the speaker. Next, you check X for the $80 value that separates the geiger effects from the long sweeps. If you have a geiger burst, you exit. For a sweep, you continue.

You continue this for X half-cycles to generate one "step" of your sweep. Then you decrement Y to go on to the next sweep step. Do this till you have completed the last step. Note that the last step is the shortest and the highest in pitch.

That should complete one sweep for you. Decrement TRPCNT4. If more sweeps are needed, then repeat the process for as many sweeps as you want. Finally, restore all the registers and exit.

There is an "oldfangled" classic cell animation demo on the companion diskette named ENGINE that you simply will not believe the first time you see and hear it. ENGINE uses the obnoxious sounds subroutine. It also has two secret ingredients called David W. Meyer, Sr., and David W. Meyer, Jr. Who, together, form one of the most fantastic father and son Apple animation teams I've ever run across anywhere, ever. And, yes, they do custom work. See the Appendix for an address.

We'll show you a simpler demo of the obnoxious sounds here and now. DEMO4 just goes through all sixteen of the sounds in order and gives you a time delay between effects.

DEMO4 produces an earth-shattering explosion a second or so after the time bomb countdown is complete. Be sure to remove all china, Ming vases, etc. from a thousand-foot radius of your Apple before running this demo.

MIND BENDERS

—Change the code so you sweep down rather than up. Do you like this?

—Extend the code so you can control pitch separately from step duration.

—What can you do with a pair of sweeps that interact with each other?

—Add suitable graphics to the time bomb.

—Which obnoxious sounds are used how in ENGINE? How is flawless animation and thick sound achieved at the same time?

—How is the frog's voice produced in RIBBIT?

PROGRAM RM-4
OBNOXIOUS SOUNDS

```
----- NEXT OBJECT FILE NAME IS OBNOXIOUS SOUNDS
6800:              ORG  $6800     ; PUT MODULE #4 AT $6800


6800:          ;  ****************************************
6800:          ;  *                                      *
6800:          ;  *       -< OBNOXIOUS  SOUNDS >-        *
6800:          ;  *                                      *
6800:          ;  *        (CUSTOM CODING METHOD)        *
6800:          ;  *                                      *
6800:          ;  *     VERSION 1.0  ($6800-$687F)       *
6800:          ;  *                                      *
6800:          ;  *              11-24-82                *
6800:          ;  *......................................*
6800:          ;  *                                      *
6800:          ;  *         COPYRIGHT C 1982 BY          *
6800:          ;  *                                      *
6800:          ;  *     DON LANCASTER AND SYNERGETICS    *
6800:          ;  *     BOX 1300, THATCHER AZ., 85552    *
6800:          ;  *                                      *
6800:          ;  *     ALL COMMERCIAL RIGHTS RESERVED   *
6800:          ;  *                                      *
6800:          ;  ****************************************


6800:          ;          *** WHAT IT DOES ***

6800:          ;  THIS MODULE GENERATES SIXTEEN DIFFERENT SOUND EFFECTS
6800:          ;  FOR USE INSIDE ANOTHER PROGRAM.
6800:          ;
6800:          ;
6800:          ;
6800:          ;


6800:          ;          *** HOW TO USE IT ***

6800:          ;  TO USE FROM MACHINE LANGUAGE, LOAD THE X REGISTER WITH
6800:          ;  A SOUND SELECTION FROM $00 TO $1F AND THEN JSR TO $6824.
6800:          ;
6800:          ;  TO USE FROM APPLESOFT, POKE 26659 WITH THE SOUND
6800:          ;  EFFECT FROM 0-15 AND CALL 26658.
6800:          ;
```

PROGRAM RM-4, CONT'D . . .

```
6800:       ;           *** GOTCHAS ***

6800:       ;   THE X REGISTER IS DESTROYED BY THIS SUBROUTINE.
6800:       ;   REGISTERS P,Y, AND A ARE SAVED FOR YOU.
6800:       ;
6800:       ;   THE PROGRAM MUST BE PLACED IN A PROTECTED AREA
6800:       ;   IF IT IS TO BE USED BY EITHER BASIC.
6800:       ;


6800:       ;           *** ENHANCEMENTS ***

6800:       ;   YOU CAN CHANGE THE EFFECTS BY CHANGING THE TRIP AND
6800:       ;   SWEEP VALUES FOR EACH FILE SELECTION.  SEE THE
6800:       ;   EFFECT FILE LISTING FOR PRESENTLY AVAILABLE EFFECTS.
6800:       ;
6800:       ;   EXTRA TONES ARE EASILY ADDED BY LENGTHENING THE SOUND
6800:       ;   EFFECT FILES SEF0-SEF15 AND CHANGING FLNGTH4


6800:       ;           *** RANDOM COMMENTS ***

6800:       ;   TO ACTIVATE THE DEMO PROGRAM THAT PLAYS ALL SIXTEEN
6800:       ;   NOTES IN ORDER, USE JSR $6800 OR CALL 26624.
6800:       ;
6800:       ;
6800:       ;
6800:       ;
```

PROGRAM RM-4, CONT'D . . .

```
6800:              ;           *** HOOKS ***

FC58:              HOME    EQU  $FC58      ; CLEAR SCREEN
FB2F:              INIT    EQU  $FB2F      ; HOME CURSOR
C030:              SPKR    EQU  $C030      ; SPEAKER CLICK OUTPUT
FCA8:              WAIT    EQU  $FCA8      ; TIME DELAY SET BY ACCUMULATOR




6800:              ;             *** DEMO  ***
6800:              ;
6800:              ;

6800:              ;       THE DEMO PROGRAM PLAYS EACH OF THE SIXTEEN
6800:              ;       SOUND EFFECTS IN ORDER, SEPARATED BY A
6800:              ;       TIME DELAY.
6800:              ;
6800:              ;
6800:              ;

6800:20 2F FB      DEMO4   JSR  INIT       ; MAKE SCREEN BLANK
6803:20 58 FC              JSR  HOME       ;
6806:A9 00                 LDA  #$00       ; START WITH FIRST NOTE
6808:48                    PHA             ;  AND SAVE ON STACK

6809:AA            NXTNOT4 TAX             ;
680A:20 24 68              JSR  OBNOX4     ; AND PLAY IT
680D:A0 0A                 LDY  #10        ; STALL FOR TIME

680F:20 A8 FC      STALL4  JSR  WAIT       ;
6812:88                    DEY             ;
6813:D0 FA                 BNE  STALL4     ; TILL DELAY DONE

6815:68                    PLA             ; GET NOTE NUMBER
6816:CD 5F 68              CMP  FLNGTH4    ; DONE WITH LAST NOTE?
6819:F0 06                 BEQ  DONE4      ;  YES, EXIT

681B:18                    CLC             ;
681C:69 01                 ADC  #$01       ;  NO, PICK NEXT NOTE
681E:48                    PHA             ;
681F:D0 E8                 BNE  NXTNOT4    ;  ALWAYS

6821:60            DONE4   RTS             ; AND EXIT
```

PROGRAM RM-4, CONT'D . . .

```
6822:                ;            *** OBNOX MODULE ***
6822:                ;
6822:                ;

6822:                ;          THIS MODULE GENERATES THE SOUND EFFECTS IN
6822:                ;          EXCHANGE FOR AN X VALUE FROM $00 TO $0F.
6822:                ;
6822:                ;
6822:                ;
6822:                ;

6822:A2 00       BASENT4 LDX  #$00        ; BASIC POKE HERE+1
6824:08          OBNOX4  PHP              ; ML ENTRY POINT
6825:48                  PHA              ;
6826:98                  TYA              ; SAVE P,A, AND Y REGS
6827:48                  PHA              ;

6828:8A                  TXA              ; RANGE CHECK ON SELECTION
6829:CD 5F 68            CMP  FLNGTH4     ;  TO MAKE SURE ITS IN FILE
682C:90 02               BCC  LOK4        ;
682E:A9 00               LDA  #$00        ; DEFAULT TO ZERO SELECTION
6830:0A          LOK4    ASLA             ;  AND DOUBLE FILE POINTER
6831:AA                  TAX              ;
6832:BD 60 68            LDA  SEF0,X      ; GET NUMBER OF TRIPS
6835:8D 5E 68            STA  TRPCNT4     ;  AND SAVE
6838:E8                  INX              ;
6839:BD 60 68            LDA  SEF0,X      ; GET SWEEP RANGE
683C:8D 40 68            STA  SWEEP4+1    ;  AND SAVE

683F:A0 00       SWEEP4  LDY  #$00        ; SWEEP VALUE POKED HERE
6841:98          NXTSWP4 TYA              ;
6842:AA                  TAX              ; DURATION
6843:98          NXTCYC4 TYA              ; PITCH
6844:20 A8 FC            JSR  WAIT        ;
6847:2C 30 C0            BIT  SPKR        ; WHAP SPEAKER
684A:E0 80               CPX  #$80        ; BYPASS IF GEIGER
684C:F0 0B               BEQ  EXIT4       ;  SPECIAL EFFECT
684E:CA                  DEX              ;
684F:D0 F2               BNE  NXTCYC4     ; ANOTHER CYCLE
6851:88                  DEY
6852:D0 ED               BNE  NXTSWP4     ; GO UP IN PITCH
6854:CE 5E 68            DEC  TRPCNT4     ; MADE ALL TRIPS?

6857:D0 E6               BNE  SWEEP4      ;  NO, REPEAT

6859:68          EXIT4   PLA              ; RESTORE REGISTERS
685A:A8                  TAY              ;
685B:68                  PLA              ;
685C:28                  PLP              ;
685D:60                  RTS              ; AND EXIT
```

PROGRAM RM-4, CONT'D . . .

```
685E:            ;             *** STASH ***


685E:01          TRPCNT4 DFB  $01        ; TRIP COUNT DECREMENTED HERE
685F:10          FLNGTH4 DFB  $10        ; SIXTEEN AVAILABLE SOUNDS



6860:            ;               *** SOUND EFFECT FILES ***


6860:            ;      EACH NOTE TAKES A TRIP AND A SWEEP VALUE IN SEQUENCE.
6860:            ;
6860:            ;      ADD $80 TO NUMBER OF GEIGER CLICKS WANTED.
6860:            ;
6860:            ;
6860:            ;


6860:01 08       SEF0    DFB  $01,$08    ; TICK
6862:01 18       SEF1    DFB  $01,$18    ; WHOPIDOOP
6864:FF 01       SEF2    DFB  $FF,$01    ; PIP
6866:06 10       SEF3    DFB  $06,$10    ; PHASOR
6868:01 30       SEF4    DFB  $01,$30    ; MUSIC SCALE
686A:20 06       SEF5    DFB  $20,$06    ; SHORT BRASS
686C:70 06       SEF6    DFB  $70,$06    ; MEDIUM BRASS
686E:FF 06       SEF7    DFB  $FF,$06    ; LONG BRASS
6870:01 A0       SEF8    DFB  $01,$A0    ; GEIGER
6872:FF 02       SEF9    DFB  $FF,$02    ; GLEEP
6874:04 1C       SEF10   DFB  $04,$1C    ; GLISSADE
6876:01 10       SEF11   DFB  $01,$10    ; QWIP
6878:30 0B       SEF12   DFB  $30,$0B    ; OBOE
687A:30 07       SEF13   DFB  $30,$07    ; FRENCH HORN
687C:50 09       SEF14   DFB  $50,$09    ; ENGLISH HORN
687E:01 64       SEF15   DFB  $01,$64    ; TIME BOMB




*** SUCCESSFUL ASSEMBLY: NO ERRORS
```

5

MUSICAL SONGS

an upgrade of the original "red book tones" song and music maker

Come on, kiddies. If you are going to reinvent the wheel, please make the thing roughly circular and put an axle somewhere near the middle, preferably pointing in some more or less reasonable direction.

The wheel in this case is a music machine that easily and simply gives you an audio tone in exchange for pitch and duration values. There are so many utterly atrocious attempts at this that it is no longer even funny.

In particular . . .

A music making subroutine MUST have totally separate and totally isolated ways of entering pitch and duration.

ANYTHING ELSE ISN'T EVEN WRONG!

If the duration of your note changes when you change the pitch, your music maker is less than worthless. Flush it.

It turns out that a really great music making subroutine has existed since year one that uniquely solves the pitch and duration interaction problem. The sub is called the *red book tones,* Woz wrote it, and it appears, of all places, in the original red book.

The red book tones are a "middleweight" technique that lets you create reasonable sounding monophonic music, as well as providing an easy way to pick up lots of different cue and prompt tones for other program uses. The original code, as it first appeared, was all of twenty-one bytes long!

Today, of course, you cannot write commercial software and get away with monophonic, fixed timbre, or constant volume sound effects. Use of multiple voices, variable volume, and duty-cycling is absolutely mandatory. But, just as LORES is an essential stepping stone to commercially useful graphics, the red book tones are a necessary learning experience along the way to top-notch musical effects.

While we will not be reinventing the wheel, we are going to add a hubcap, some chrome, and better bearings.

First, we all call the newer version REDTONE. As with the original, it gives you a constant frequency square-wave tone in exchange for pitch and duration values. We've put REDTONE into source code so you can relocate it anywhere you want. The original code sat on page zero and had some Applesloth compatibility problems. The obvious choice of page three is so overloaded these days, that it is best to have something you can put anywhere you want.

REDTONE saves all the working registers to avoid conflicts with your high level code. The pitch and duration values are also saved for you, so you needn't reload the same duration value over and over again for cues or prompts.

There is now a *silent* pitch value of $FF. This is most handy for rests and pauses. The silence is timed out to the same duration value any other note would be. As a convenience, the notes are echoed to the cassette output port. You can greatly improve the sound by going through a small hi-fi amplifier and larger speaker. Use standard audio cables.

The maximum duration on the original code was a little short, particularly when it came to playing whole notes at low tempos, so REDTONE has a feature called a *duration multiplier* that lets you extend the duration in binary mulitples. You now have all the duration range you could possibly ever use, plus a ridiculous bunch more.

And that just about covers the code improvements. We've also made two use improvements. The first involves better pitch accuracy, and the second lets your Assembler enter music in a sane and more or less musical way. For instance, a half note of middle C is entered as "C1,H,". There are no worries about funny numbers.

Tempo is presently set by changing a single value before assembly. You can easily upgrade to a "real time" tempo control. I've purposely left this as an "exercise for the student."

Pitch Accuracy

Most people try to set up a tone generator to make some certain pitch exactly hit some musical note. Then they go up and down the scale from there, trying to "fit" the notes to the 8-bit pitch values needed.

The problem is that this technique works well for some notes and poorly for others. Some notes just won't fit and will sound out of tune.

Some review. An *octave* is a 2:1 frequency change, and is just about as far as you can easily reach on a piano, say from middle C to the next higher C. People have messed with how many notes go where for a long time, but today, most everyone uses a compromise system called the *equally tempered* scale.

The equally tempered scale has twelve notes per octave. The notes in the "key" of C are called, C, C#, D, E, F, F#, G, G#, A, A#, B, and back again to the next C that's one octave higher. Note that there is no "E#" or "B#" as such. Other keys may name these notes differently and may start at a different point, but regardless of which key is in use, there are only twelve notes per octave.

The pitch of a note is related to that note's frequency, which is called out in hertz, or cycles per second. For instance, the pitch of the A above middle C is standardized to a frequency of 440 Hz.

Since the ear is a logarithmic type device, it expects low frequency differences between notes for the low notes, and high frequency differences between notes for the high notes. If you tried to create a linear "scale" that went, say 300, 350, 400, 450, 500, . . . etc. Hz, it would sound very weird indeed.

Unmusical, even.

To get a log spacing of 12 notes over one octave, each successive *equally tempered* note has to be the *twelfth root of two* higher in frequency. This is roughly a factor of 1.06. Each note ends up roughly 6 percent higher in frequency than its neighbor.

The interval from note to note is called a *semitone.* A semitone is the difference from one key to the immediate next one on a piano, regardless of key color. A semitone is also a 6 percent increase in frequency. A pitch change of one semitone is thus only a few hertz for low notes, but is very much more than this for high notes.

How accurate do the tones have to be? It turns out that very few people have what is called "absolute pitch," so if the whole song is uniformly mistuned too high or too low, nobody will be able to tell.

What counts is the relation between the notes, or "relative pitch," and here, things get sticky fast . . .

Few people can tell ABSOLUTE PITCH, so it really doesn't matter whether all the notes are exactly set to their intended absolute frequencies.

Just about anybody can tell RELATIVE PITCH, so it is super important that the notes all sound good together.

Thus, if an "A" is really 480 Hz rather than 440, the odds are high that nobody will notice on a stand-alone song. So long, of course, that all the *other* notes are equally offset from where they belong *by the same proportion.* What is critical is the *relative* frequency difference between "A" and "A#," or between any other notes.

How critical is critical? Musicians call one one-hundredth of a semitone a *cent.* A one cent frequency error is an error of just under 0.06 percent in the ratio of two notes. It turns out that the best musicians can just barely spot a one cent frequency error, while an average careful listener can spot a three cent error.

The trick is to get accurate relative notes consistent with a pitch word that is only 8 bits wide. If you just force any old note to be exact and then try to find magic values for the other notes, one or more of them will sound sour.

If you play with funny numbers long enough, you'll find that there is a little known but super important *series* of 8-bit pitch values that give far and away the most accurate notes you can possibly get using 8-bit values. Any other attempt at pitch values will fall short of this optimum series, and you'll get several sour notes.

Here's the magic series and the notes involved . . .

"MAGIC" 8-BIT PITCH VALUES

232	(A)
219	(A#)
207	(B)
195	(C)
184	(C#)
174	(D)
164	(D#)
155	(E)
146	(F)
138	(F#)
130	(G)
123	(G#)
116	(A)

These notes are all accurate to better than three cents in relative pitch. Once again, this is a "magic" series. Any other choice of pitch values will give you at least one sour note. Note that the pitch values will set the time between speaker motions of REDTONE, so the *higher* the pitch value, the *lower* the pitch or frequency of the note you get. It takes two shoves, one forward and one backward, of the speaker cone, to generate one full cycle of a REDTONE square wave. The timbre you get is a "woody" one roughly akin to a clarinet or a stopped organ pipe.

The approximate notes you actually get with REDTONE are shown in parentheses. You can continue up in pitch, but you'll eventually pick up some sour notes on the way. Just divide each of the "magic" values by two for the next octave, and so on.

Note that you will only have seven or fewer bits of accuracy for these higher notes. Which means a few of them may be off in pitch. By the way, you also have an additional magic 8-bit pitch value of 246. This translates to a REDTONE "G#" or "A*b*," and it seemed to make more sense to start at "A" instead.

Separating Pitch and Duration

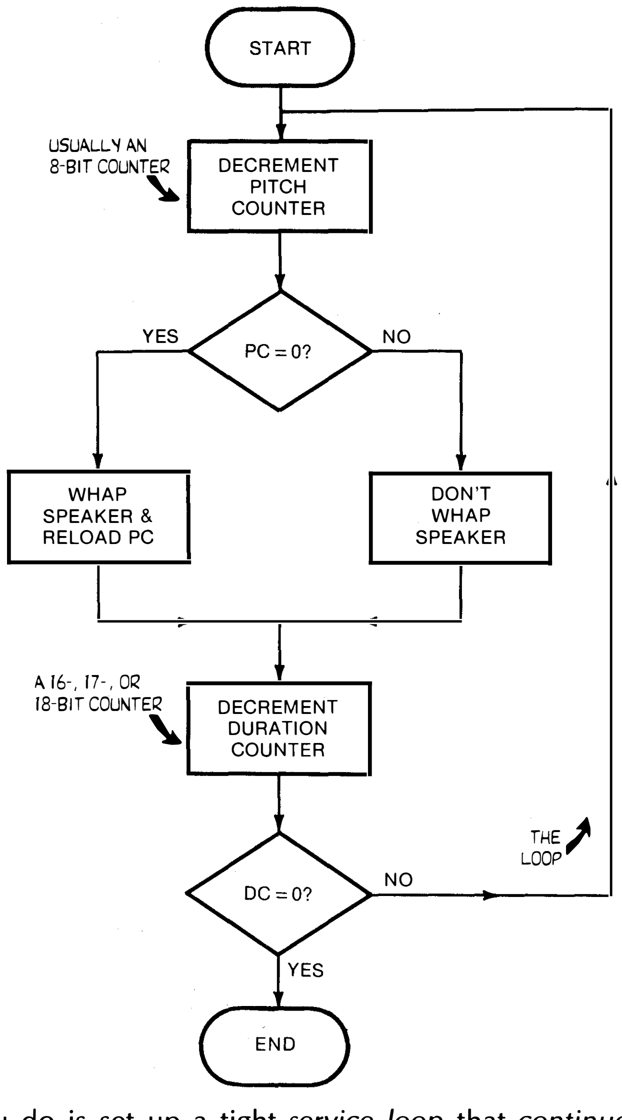
The "obvious" way to generate a tone is to count one register down to get the pitch. Each completed countdown whaps the speaker once. To get duration, you then count the total number of whappings.

Which is simple but wrong.

The trouble is that the high notes will sound much shorter than the low notes. Which gets to be a real mess. Any decent music maker subroutine *must* separate pitch and duration.

Here's how to do it . . .

USING A "SERVICE" LOOP TO SEPARATE PITCH & DURATION:



What you do is set up a tight *service loop* that *continuously* tests *both* the pitch and duration values. Two counters are involved, an 8-bit pitch counter, and a 16-bit or longer duration counter. The service loop continuously decrements *both* of these counters. When the magic pitch value is hit, the speaker gets whapped. When the magic duration value is hit, the tone ends. Since the duration values are usually much larger than the pitch values, you will normally get many pitch cycles in your note.

The way the original red book tones got its 16-bit duration values was to take an 8-bit duration value and multiply it by 256 using the Y register. Thus, the Y register had to go all the way around for each count of the duration counter.

All of which elegantly solved keeping pitch and duration separate.

A Duration Multiplier

The only little problem with this scheme was that 16 bits worth of duration weren't quite enough for some uses. Things were OK for simple songs, but for dotted half notes or for whole notes played at slow tempos, there simply wasn't enough duration to fully sound the note. The maximum duration was just under one second.

REDTONE gets around this by going to as many as 24 bits for the duration counter. It turns out that REDTONE never needs the accumulator, so this register is free to be used as a multiplying counter.

Here's how it works. You always initialize the accumulator to $00. Now, say you add some magic value to the accumulator and test for zero. The results you get depend on what you add. Four useful results include . . .

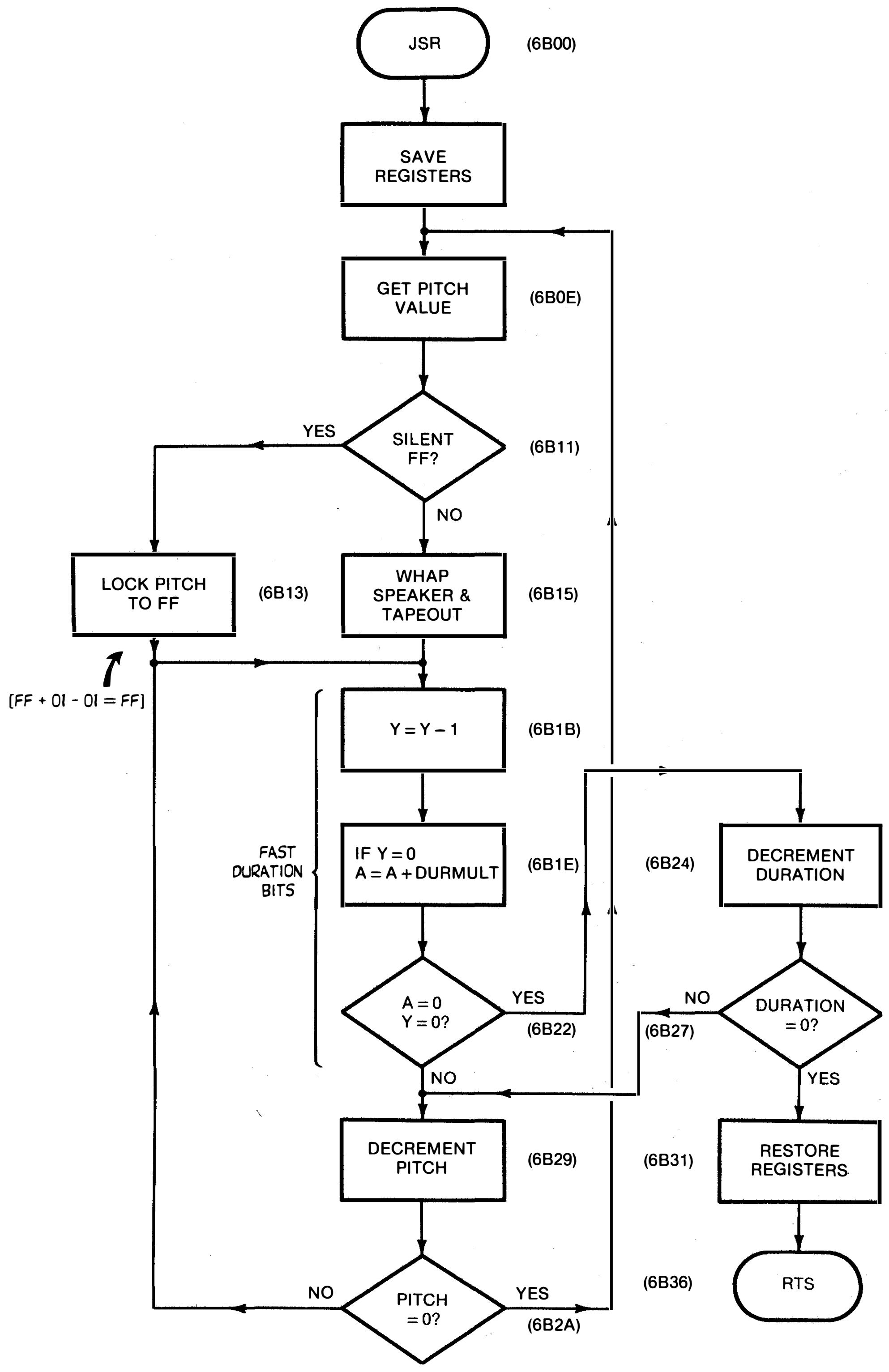
```
Adding $00 gives you

                00 00 00 00 00 00 00 00 00

                                  and multiplies by ONE.

Adding $80 gives you

                00 80 00 80 00 80 00 80 00

                                  and multiplies by TWO.

Adding $40 gives you

                00 40 80 C0 00 40 80 C0 00

                                  and multiplies by FOUR.

Adding $20 gives you

               00 20 40 60 80 A0 C0 E0 00

                                  and multiplies by EIGHT.
```

What you do is count down the duration counter every time you get a zero result. Thus, the $40 adder only decrements the duration counter on every *fourth* trip through the service loop. This makes the note last four times longer.

Usually, a "X2" multiplier is just what you need for most music. You can go up to "X256" multiplication, using an $01 magic value, if you want to get ridiculous.

With these details out of the way, let's look at the REDTONE subroutine. Here's the flowchart . . .

REDTONE FLOWCHART:



You enter REDTONE with a pitch value of PITCH5 and a duration value of DURAT5. These values are not destroyed should you want to reuse them for simple prompts. You must also have a multiplier value in DURMULT5, but leaving this value at $80 will give good results for most uses.

The registers are first saved. Then DURAT5 is copied into DURCNT5 where it can be counted down. The copying saves you having to reenter the same duration each time for simple prompts. This is followed by clearing the Y register and the accumulator. The accumulator will be used for the duration 1-2-4 multiplier, while the Y register will be used to scale the duration by 256. You can alternately use the Y register to adjust tempo in real time.

The pitch value is placed in the X register and tested. If the pitch is not $FF, the note is accepted and processed as usual. The speaker is then whapped, and then is echoed to the cassette output.

Next, the service loop takes over. First, the Y register is decremented. If Y hits zero, then the duration multiplier gets activated, by adding the multiplier value and testing for a zero result. If the Y register has gone all the way around and if the duration multiplier gives you a zero result, then, and only then, is DURCNT5 decremented. Note that this has the effect of multiplying the DURCNT5 value first by 256 and then by the accumulator multiplier of 1, 2, 4, or whatever.

If we have not gotten a zero duration value, we then knock one off the pitch counter and repeat the service loop process. The speaker gets whapped only on zero values of the pitch counter. Thus a single service loop separately keeps track of pitch and duration with only negligible interaction.

Note that the duration is the product of three 8-bit values. Duration is set by multiplying the Y register *times* DURAT5 *times* DURMULT5.

Every pitch zero, the speaker gets hit, and the X register gets reloaded with a new pitch value. The only exit from all this happens when DURCNT5 finally hits zero. At that point, the registers are restored and the subroutine exits to your calling code.

One final detail. If your chosen pitch value is $FF, the speaker is not sounded. This gives you a silent note, a pause, or a rest.

The side loop at LOCKX handles this detail for you. IF the pitch is $FF, the pitch is incremented to $00 by LOCKX, and then later decremented back to $FF in the main service loop. Thus a $FF pitch value *stays* at $FF all the way through the duration timing. This happens because $FF + $01 − $01 = $FF. A sounding pitch value gets counted down to zero and whaps the speaker every trip. A silent pitch value stays at $FF and bypasses the speaker, producing no sound.

A Demo or Two

The SONGPLY demo exercises REDTONE for you, playing that ever favorite song that Tarzan used to sing during his zebra maintenance days. SONGPLY works by picking pitch and duration value out of a songfile called TARZAN.

We have used a 16-bit full wide pointer to access the song file, so you can have more than 128 notes total in your song. This pointer is called NOTEP and is page zero stashed at $EF and $F0. To link SONGPLY to different songs, you change these pointers as needed.

The pause generator inside SONGPLY gives you a brief pause between notes that is set by PAUSE. Experiment to get the best results. The minimum PAUSE value is $01. Do not use $00!

Should your particular song demand some notes that slur or tie together, just use a minimum value of PAUSE, say $01. Then use six-teenths rests or whatever between those notes that do not slur.

Be sure to study the source code on SONGPLY very carefully, for it shows you a fairly friendly way to use labels to simplify writing your own songs. We'll leave details on this for you to puzzle out.

One tip. Use two $00 for END values at the end of your note file. That way, should you have an error in your list, you will still stop, catching the second END value.

Where to from here? We have thrown in a quick tester called the TIMBRE TESTER that will get you started in experimenting with different "voices" for your Apple. To use the TIMBRE TESTER, just put a number series into TIMBFLE and the number of numbers into TFLEN. You can get the magic numbers by trial and error, from a venture into Fourier Series (gulp!), or from full-fledged duty cycling experiments.

TIMBRE TESTER works by generating a waveform with many possible zero crossings. As you change the number of zero crossings and the spacing between them, the harmonic content of the note changes, giving you different "voicing" for your Apple.

As examples, a waveform that has very strong fourth, fifth, and sixth harmonics, with a very weak fundamental, second, and third, will sound as a three note major triad chord. Other pleasant two note effects include a strong second and third, third and fourth, fourth and fifth, second and fifth, and third and fifth harmonics. A note with no low harmonics except for the fundamental will voice as a pure and flute-like sinewave.

You can get these harmonics the way you want them, either by trial and error, or else by fancy math.

With duty cycling, you use lots of very high frequency cycles, set up so that the *average* speaker cone position matches the waveform you are trying to generate. You can easily get pure sinewaves, variable volume, and even exceptionally good human voice synthesis with fancy enough duty cycling.

Each value in TIMBFLE specs the delay time in microseconds, multiplied by five, between cone whappings. For a "fat" sound, you whap the cone many times per frequency cycle. As a fine point, knock two off each VOICE5 value, except for the last one. Knock four off it. Why?

The TIMBRE TESTER can easily give you string and woodwind-style tones, flutelike sinewaves, bells, two notes at once, three notes at once, "noisy" sounds, and even voice. All it takes is the right numbers in the right order.

Finding them is half the fun.

MIND BENDERS

—Extend TARZAN by entering the "hard parts" that I left out.

—Write your own songs for SONGPLY.

—Modify SONGPLY so you can control the tempo from a game paddle. Hint: Put the tempo into the Y register.

—Examine the *exact* timing involved in REDTONE. What effects do slight variations from "perfect" loop timing have?

—Show why LOCKX is not needed and how to replace it.

—Play "Applesoft" by using $D000 as NOTEFL. Why are the results no longer equally tempered?

—Modify REDTONE for a string voice with an 8:1 duty cycle.

—Show a two byte change to SONGPLY that lets you edit by playing one note at a time.

—Use the TIMBRE TESTER to produce a pure sinewave, and then two notes at once. Then, ring a bell.

—Some poor attempts at duty cycling may buzz or whine. Why? How can you eliminate this?

PROGRAM RM-5
MUSICAL SONGS

```
----- NEXT OBJECT FILE NAME IS MUSICAL SONGS
6900:                 ORG  $6900     ; PUT MODULE #5 AT $6900


6900:          ;   *****************************************
6900:          ;   *                                       *
6900:          ;   *        -< MUSICAL SONGS >-            *
6900:          ;   *                                       *
6900:          ;   *     ( MODIFIED RED BOOK TONES )       *
6900:          ;   *                                       *
6900:          ;   *      VERSION 1.0  ($6900-$6B3A)       *
6900:          ;   *                                       *
6900:          ;   *              5-24-83                  *
6900:          ;   *.......................................*
6900:          ;   *                                       *
6900:          ;   *         COPYRIGHT C 1983 BY           *
6900:          ;   *                                       *
6900:          ;   *     DON LANCASTER AND SYNERGETICS     *
6900:          ;   *     BOX 1300, THATCHER AZ., 85552     *
6900:          ;   *                                       *
6900:          ;   *     ALL COMMERCIAL RIGHTS RESERVED    *
6900:          ;   *                                       *
6900:          ;   *****************************************


6900:          ;          *** WHAT IT DOES ***

6900:          ;   THIS MODULE SHOWS YOU HOW TO USE THE MODIFIED
6900:          ;   RED BOOK TONE SUBROUTINE TO PLAY MUSICAL SONGS.
6900:          ;
6900:          ;   THERE IS ALSO A TIMBER TESTER FOR EVALUATION
6900:          ;   OF SPECIAL APPLE VOICES AND SOUND EFFECTS.


6900:          ;          *** HOW TO USE IT ***

6900:          ;   TO PLAY A SINGLE NOTE:
6900:          ;      PUT YOUR PITCH IN PITCH5 AT $6B3A (27450).
6900:          ;      PUT THE DURATION IN DURAT5 AT $6B37 (27447).
6900:          ;      THEN JSR REDTONE AT $6B00 (27392).

6900:          ;   TO PLAY YOUR OWN SONG:
6900:          ;      PUT THE STARTING ADDRESS OF YOUR SONG INTO
6900:          ;      SONGLOC AND SONGLOC+1 AT $6944 AND $6945.
6900:          ;      THEN JSR SONGPLY AT $6910.  APPLESLOTH
6900:          ;      EQUIVALENTS ARE 26948, 26949, AND 26896.

6900:          ;   TO PLAY TARZAN:
6900:          ;      DO A JSR $6900 OR CALL 26880.
```

PROGRAM RM-5, CONT'D . . .

```
6900:           ;             *** GOTCHAS ***

6900:           ;   A CHANGE OF TEMPO PRESENTLY NEEDS REASSEMBLY.
6900:           ;   REDTONE IS LIMITED TO "WOODWIND" SQUARE WAVES.


6900:           ;             *** ENHANCEMENTS ***

6900:           ;   THIS SOURCE CODE ALSO SHOWS YOU HOW TO COMPOSE
6900:           ;   YOUR OWN SONGS IN THE EQUALLY TEMPERED MUSICAL
6900:           ;   SCALE BY USING LABELS THAT SIMPLIFY NOTE ENTRY.
6900:           ;


6900:           ;             *** RANDOM COMMENTS ***

6900:           ;   THIS IS "MIDDLEWEIGHT" CODE INTENDED TO SHOW
6900:           ;   PROGRAMMING SKILLS AND TECHNIQUES.  FANCIER
6900:           ;   METHODS SHOULD BE USED FOR COMMERCIAL PROGRAMS
6900:           ;   OR FOR SERIOUS MUSICAL COMPOSITION.
6900:           ;



6900:           ;                  *** HOOKS ***


FC58:           HOME    EQU  $FC58     ; CLEAR TEXT SCREEN AND HOME CURSOR
FB2F:           INIT    EQU  $FB2F     ; INITIALIZE TEXT SCREEN
C010:           KBDSTR  EQU  $C010     ; KEYBOARD STROBE
C000:           IOADR   EQU  $C000     ; KEYBOARD INPUT LOCATION
C030:           SPKR    EQU  $C030     ; SPEAKER CLICK OUTPUT
C020:           TAPEOUT EQU  $C020     ; CASSETTE TAPE OUT (TONE ECHO)
FCA8:           WAIT    EQU  $FCA8     ; MONITOR TIME DELAY

00EF:           NOTEP   EQU  $EF       ; NOTE POINTER PAIR FOR SONGLPLY
```

PROGRAM RM-5, CONT'D . . .

```
6900:            83 ;               *** CONSTANTS ***


6900:            85 ;       PITCH LABELS USED FOR SONG COMPOSITION
6900:            86 ;
6900:            87 ;        FOR BEST SOUND, ALWAYS USE THE NOTE
6900:            88 ;        VALUES NEAREST THE TOP OF THIS LIST.


00E8:            90 A1      EQU  232       ; NOTE A BELOW MIDDLE C

00DB:            92 A1S     EQU  219       ;      A#
00DB:            93 B1F     EQU  219       ;      Bb
00CF:            94 B1      EQU  207       ;      B
00C3:            95 C1      EQU  195       ;      C
00B8:            96 C1S     EQU  184       ;      C#
00B8:            97 D1F     EQU  184       ;      Db
00AE:            98 D1      EQU  174       ;      D
00A4:            99 D1S     EQU  164       ;      D#
00A4:           100 E1F     EQU  164       ;      Eb
009B:           101 E1      EQU  155       ;      E
0092:           102 F1      EQU  146       ;      F
008A:           103 F1S     EQU  138       ;      F#
008A:           104 G1F     EQU  138       ;      Gb
0082:           105 G1      EQU  130       ;      G
007B:           106 G1S     EQU  123       ;      G#
007B:           107 A1F     EQU  123       ;      Ab

0074:           109 A2      EQU  116       ; NOTE A ABOVE MIDDLE C

006E:           111 A2S     EQU  110       ;      A#
006E:           112 B2F     EQU  110       ;      Bb
0067:           113 B2      EQU  103       ;      B
0062:           114 C2      EQU  98        ;      C
005C:           115 C2S     EQU  92        ;      C#
005C:           116 D2F     EQU  92        ;      Db
0057:           117 D2      EQU  87        ;      D
0052:           118 D2S     EQU  82        ;      D#
0052:           119 E2F     EQU  82        ;      Eb
004E:           120 E2      EQU  78        ;      E
0049:           121 F2      EQU  73        ;      F
0045:           122 F2S     EQU  69        ;      F#
0045:           123 G2F     EQU  69        ;      Gb
0041:           124 G2      EQU  65        ;      G
003D:           125 G2S     EQU  61        ;      G#
003D:           126 A2F     EQU  61        ;      Ab

003A:           128 A3      EQU  58        ; SECOND A ABOVE MIDDLE C


00FF:           130 R       EQU  $FF       ; SILENT OR REST
0000:           131 END     EQU  $00       ; END OF SONG  (USE TWICE!)
```

PROGRAM RM-5, CONT'D . . .

```
6900:          ;      DURATION LABELS USED FOR SONG COMPOSITION


6900:          ;         A REPEAT ASSEMBLY PASS IS NEEDED AT
6900:          ;         PRESENT FOR EACH CHANGE IN TEMPO.


0009:            TEMPO   EQU  $09        ; MASTER TEMPO CONTROL  ($0F MAXIMUM!)
0080:            MULT    EQU  $80        ; TEMPO MULTIPLIER (00=X1 $80=X2 $40=X4
0048:            PAUSE   EQU  $48        ; INTERNOTE PAUSE TIME

0009:            S       EQU  TEMPO*1    ; SIXTEENTH NOTE
000D:            DS      EQU  S/2+S      ; DOTTED SIXTEENTH
0012:            E       EQU  TEMPO*2    ; EIGHTH NOTE
001B:            DE      EQU  TEMPO*3    ; DOTTED EIGHTH
0024:            Q       EQU  TEMPO*4    ; QUARTER NOTE
0036:            DQ      EQU  TEMPO*6    ; DOTTED QUARTER
0048:            H       EQU  TEMPO*8    ; HALF NOTE
006C:            DH      EQU  TEMPO*12   ; DOTTED HALF NOTE
0090:            W       EQU  TEMPO*16   ; WHOLE NOTE
```

PROGRAM RM-5, CONT'D . . .

```
6900:             ;        ***  MUSICAL SONGS  ***
6900:             ;
6900:             ;   THIS SUBROUTINE USES REDTONE TO PLAY THE
6900:             ;   SONG WHOSE STARTING ADDRESS IS IN SONGLOC.


6900:20 2F FB     TAR     JSR  INIT        ; INITIALIZE TEXT SCREEN
6903:20 58 FC             JSR  HOME        ;  AND CLEAR IT

6906:A9 46                LDA  #>TARZAN    ; TO PLAY TARZAN ONLY
6908:8D 44 69             STA  SONGLOC     ;
690B:A9 69                LDA  #<TARZAN    ;
690D:8D 45 69             STA  SONGLOC+1   ;


6910:AD 44 69     SONGPLY LDA  SONGLOC     ; MOVE SONG ADDRESS TO POINTER
6913:85 EF                STA  NOTEP       ;
6915:AD 45 69             LDA  SONGLOC+1   ; POSITION THEN PAGE AS USUAL
6918:85 F0                STA  NOTEP+1     ;

691A:A0 00                LDY  #$00        ; FOR PURE INDIRECT
691C:A9 80                LDA  #MULT       ; SET DURATION MULTIPLIER
691E:8D 39 6B             STA  DURMUL5     ; AND POKE TO REDTONE

6921:B1 EF        MORE5   LDA  (NOTEP),Y   ; GET PITCH VALUE
6923:F0 1E                BEQ  DONE5       ; EXIT IF END
6925:8D 3A 6B             STA  PITCH5      ; POKE PITCH
6928:E6 EF                INC  NOTEP       ; GO TO NEXT FILE VALUE
692A:D0 02                BNE  NOCY5       ;  PAGE OVERFLOW?
692C:E6 F0                INC  NOTEP+1     ;   YES
692E:B1 EF        NOCY5   LDA  (NOTEP),Y   ; GET DURATION VALUE
6930:8D 37 6B             STA  DURAT5      ; STASH DURATION VALUE
6933:20 00 6B             JSR  REDTONE     ; PLAY THE NOTE
6936:A9 48                LDA  #PAUSE      ; GET INTERNOTE DELAY
6938:20 A8 FC             JSR  WAIT        ; AND DELAY
693B:E6 EF                INC  NOTEP       ; GO TO NEXT FILE VALUE
693D:D0 E2                BNE  MORE5       ;  PAGE OVERFLOW?
693F:E6 F0                INC  NOTEP+1     ;   YES
6941:D0 DE                BNE  MORE5       ; ALWAYS (WELL, ALMOST!)

6943:60           DONE5   RTS              ; END OF SONG
```

PROGRAM RM-5, CONT'D . . .

```
6944:           196 ;          *** SONG POINTER STASE ***

6944:46 69      198 SONGLOC DFB  >TARZAN,<TARZAN


6946:           200 ;          *** SONG FILE ***


6946:           202 ;   EACH NOTE IS ENTERED, PITCH FIRST AND
6946:           203 ;   DURATION SECOND USING LABELS AS SHOWN.
6946:           204 ;



6946:74 48 74   206 TARZAN  DFB             A2,H,A2,H,G1,Q,F1S,Q,F1S,H
6949:48 82 24
694C:8A 24 8A
694F:48
6950:92 24 8A   207         DFB             F1,Q,F1S,Q,F1S,W,R,H
6953:24 8A 90
6956:FF 48
6958:92 24 8A   208         DFB             F1,Q,F1S,Q,F1S,H,F1,Q,F1S,Q
695B:24 8A 48
695E:92 24 8A
6961:24
6962:74 48 8A   209         DFB             A2,H,F1S,DQ,A2,E,G1,W,E1,DH,R,Q
6965:36 74 12
6968:82 90 9B
696B:6C FF 24
696E:9B 48 A4   210         DFB             E1,H,D1S,Q,E1,Q,E1,H
6971:24 9B 24
6974:9B 48
6976:A4 24 9B   211         DFB             D1S,Q,E1,Q,A2,W,R,H
6979:24 74 90
697C:FF 48
697E:8A 24 9B   212         DFB             F1S,Q,E1,Q,F1S,Q,A2,DH
6981:24 8A 24
6984:74 6C
6986:67 6C 67   213         DFB             B2,DH,B2,Q,E1,W,R,H
6989:24 9B 90
698C:FF 48
698E:74 48 74   214         DFB             A2,H,A2,H,G1,Q,F1S,Q,F1S,H
6991:48 82 24
6994:8A 24 8A
6997:48
6998:92 24 8A   215         DFB             F1,Q,F1S,Q,F1S,W,R,H
699B:24 8A 90
699E:FF 48
```

PROGRAM RM-5, CONT'D . . .

```
69A0:92 24 8A          DFB          F1,Q,F1S,Q,F1S,H,F1,Q,G1,Q
69A3:24 8A 48
69A6:92 24 82
69A9:24
69AA:82 24 8A          DFB          G1,Q,F1S,Q,E1,DQ,C2S,E,E1,DH,D1,H
69AD:24 9B 36
69B0:5C 12 9B
69B3:6C AE 48
69B6:FF 24 AE          DFB          R,Q,D1,Q,D1,H,C1S,Q,D1,Q
69B9:24 AE 48
69BC:B8 24 AE
69BF:24
69C0:92 48 9B          DFB          F1,H,E1,Q,D1,Q,D2,W,R,Q
69C3:24 AE 24
69C6:57 90 FF
69C9:24
69CA:92 24 9B          DFB          F1,Q,E1,Q,F1S,Q,A2,E,R,E,D1,Q
69CD:24 8A 24
69D0:74 12 FF
69D3:12 AE 24
69D6:9B 24 8A          DFB          E1,Q,F1S,Q,A2,E,R,E
69D9:24 74 12
69DC:FF 12
69DE:E8 24 CF          DFB          A1,Q,B1,Q,F1S,Q,E1,W,D1,Q
69E1:24 8A 24
69E4:9B 90 AE
69E7:24
69E8:00 00             DFB          END,END
```

PROGRAM RM-5, CONT'D . . .

```
69EA:            ;           *** TIMBRE TESTER ***
69EA:            ;
69EA:            ;      THIS ROUTINE LETS YOU EVALUATE SPECIAL VOICES,
69EA:            ;      DUTY CYCLING, MULTI-TONES AND OTHER EFFECTS.
69EA:            ;
69EA:            ;      TO USE, LOAD TIMBFLE WITH THE DELAY VALUES
69EA:            ;      BETWEEN ZERO CROSSINGS.  LOAD TFLENGTH WITH
69EA:            ;      THE NUMBER OF ZERO CROSSINGS PER FUNDAMENTAL
69EA:            ;      NOTE CYCLE.
69EA:            ;
69EA:            ;      TO RUN:
69EA:            ;
69EA:            ;        JSR $6AC0 FROM MACHINE LANGUAGE
69EA:            ;        CALL 27328 FROM APPLESLOTH.
69EA:            ;
69EA:            ;        EXIT ON ANY KEY PRESSED.



6AC0:                     ORG  TAR+$01C0 ; LEAVE ROOM FOR SONG FILES

6AC0:2C 10 C0     TIMBRE  BIT  KBDSTR    ; RESET KEYBOARD
6AC3:AE DE 6A     RESCAN5 LDX  TFLENGTH  ; START NEW SCAN
6AC6:CA           NEXT5   DEX            ; NEXT VALUE
6AC7:30 FA                BMI  RESCAN5   ; RESET IF COMPLETE
6AC9:BC DF 6A             LDY  TIMBFLE,X ; GET DELAY VALUE
6ACC:88           LOOP5   DEY            ; DELAY 5N+1 CYCLES
6ACD:D0 FD                BNE  LOOP5     ; STALL FOR TIME
6ACF:2C 30 C0             BIT  SPKR      ; WHAP SPEAKER
6AD2:2C 20 C0             BIT  TAPEOUT   ; WHAP CASSETTE OUTPUT
6AD5:2C 00 C0             BIT  IOADR     ; KEYPRESSED?
6AD8:10 EC                BPL  NEXT5     ; REPEAT IF NO KP
6ADA:2C 10 C0             BIT  KBDSTR    ; RESET KEYSTROBE
6ADD:60                   RTS            ; AND EXIT


6ADE:            ;     *** TIMBRE DELAY VALUES ***


6ADE:08           TFLENGTH DFB $08       ; NUMBER OF CROSSINGS IN TIMBFLE


6ADF:60 6A 6A     TIMBFLE DFB             $60,$6A,$6A,$27
6AE2:27
6AE3:27 6A 6A             DFB             $27,$6A,$6A,$5E
6AE6:5E
```

PROGRAM RM-5, CONT'D . . .

```
6AE7:            ;       *** MODIFIED RED BOOK TONE SUBROUTINE ***


6AE7:            ;       A JSR TO REDTONE PLAYS A SINGLE NOTE.
6AE7:            ;
6AE7:            ;       THE PITCH MUST BE PREPLACED IN PITCH5
6AE7:            ;       DURATION MUST BE PREPLACED IN DURAT5.
6AE7:            ;
6AE7:            ;       A PITCH5 VALUE OF $FF IS SILENT.


6B00:                     ORG  TAR+$0200 ; LEAVE ROOM FOR TIMBRE FILES

6B00:48            REDTONE PHA            ; SAVE REGISTERS
6B01:98                    TYA            ;
6B02:48                    PHA            ;
6B03:8A                    TXA            ;
6B04:48                    PHA            ;
6B05:AD 37 6B              LDA  DURAT5    ; MOVE DURATION VALUE TO
6B08:8D 38 6B              STA  DURCNT5   ;   COUNTABLE LOCATION

6B0B:A0 00                 LDY  #$00      ; INIT FAST DURATION COUNTER
6B0D:98                    TYA            ; INIT DURATION MULTIPLIER

6B0E:AE 3A 6B      WHAP    LDX  PITCH5    ; GET PITCH VALUE
6B11:E0 FF                 CPX  #$FF      ; IS IT SILENT?
6B13:F0 19                 BEQ  LOCKX     ; YES, KEEP IT SILENT
6B15:2C 30 C0              BIT  SPKR      ; WHAP SPEAKER
6B18:2C 20 C0              BIT  TAPEOUT   ; AND ECHO TO CASSETTE OUTPUT
6B1B:88            NOWHAP  DEY            ; DECREMENT FAST DURATION COUNT
6B1C:D0 0B                 BNE  NOC5      ; IF NO BORROW
6B1E:18                    CLC            ;
6B1F:6D 39 6B              ADC  DURMUL5   ; DURATION MULTIPLIER
6B22:D0 05                 BNE  NOC5      ;  IGNORE ALL BUT ZERO RESULTS
6B24:CE 38 6B              DEC  DURCNT5   ; DECREMENT SLOW DURATION
6B27:F0 08                 BEQ  EXIT5     ; IF FINISHED
6B29:CA            NOC5    DEX            ; DECREMENT PITCH VALUE
6B2A:D0 EF                 BNE  NOWHAP    ; PITCH NOT DONE
6B2C:F0 E0                 BEQ  WHAP      ; PITCH DONE, ALWAYS TAKEN

6B2E:E8            LOCKX   INX            ; TRAP X TO $FF
6B2F:F0 EA                 BEQ  NOWHAP    ; ALWAYS TAKEN

6B31:68            EXIT5   PLA            ; RESTORE REGISTERS
6B32:AA                    TAX            ;
6B33:68                    PLA            ;
6B34:A8                    TAY            ;
6B35:68                    PLA            ;
6B36:60                    RTS            ; AND EXIT
```

PROGRAM RM-5, CONT'D . . .

```
6B37:              ;            *** REDTONE STASH ***

6B37:72            DURAT5  DFB  $72        ; DURATION GOES HERE
6B38:72            DURCNT5 DFB  $72        ;  GETS COUNTED HERE
6B39:80            DURMUL5 DFB  $80        ; DURATION MULTIPLIER
6B3A:72            PITCH5  DFB  $72        ; PITCH GOES HERE



*** SUCCESSFUL ASSEMBLY: NO ERRORS
```

6

OPTION PICKER

a general and flexible way to handle menu selections and in-program jumps

Just about any larger program eventually gets to a point where it has to jump six ways from Sunday. These may involve internal jumps, such as when an adventure decides it has to check to be sure the giant armadillo is awake. Or, they might involve user input, such as a menu selection, the "T" for trace command of a monitor, or the "[S]" save command of a word processor.

A code module that lets a program go to one of many possible tasks is called an *option picker* . . .

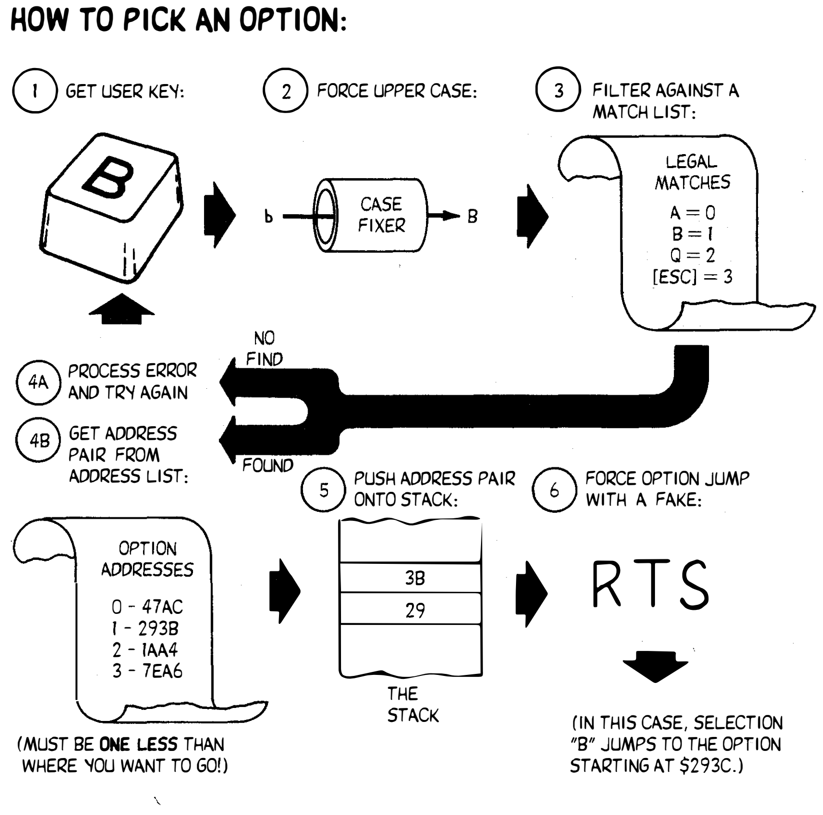
OPTION PICKER—
A code module that lets a program continue by jumping to a selected one of many possible tasks.

Now, option picking doesn't sound like a very big deal. The trick is to come up with one single option picker that you can adapt to any program you want to, while keeping things as short and as flexible as possible.

the code for each and every application. This way, you know your code works ahead of time. Any problems are likely to be file problems that are easily spotted and more easily fixed.

Let's check into the most general and often the "best" way to pick one of many options. To do this, get a character from a program or a keyboard. Then, if needed, change lowercase to uppercase. Next, filter your character by looking into a file to find a character match. If there's no match, process the error and try again. If you do find a match, go to a second file and grab an address to go to. Then, jump to that address.

Something like this . . .


HOW TO PICK AN OPTION:
1 GET USER KEY:
B
2 FORCE UPPER CASE:
b
CASE FIXER
B
3 FILTER AGAINST A MATCH LIST:
LEGAL MATCHES
A = 0
B = 1
Q = 2
[ESC] = 3
NO FIND
4A PROCESS ERROR AND TRY AGAIN
4B GET ADDRESS PAIR FROM ADDRESS LIST:
FOUND
OPTION ADDRESSES
0 - 47AC
1 - 293B
2 - 1AA4
3 - 7EA6
(MUST BE ONE LESS THAN WHERE YOU WANT TO GO!)
5 PUSH ADDRESS PAIR ONTO STACK:
3B
29
THE STACK
6 FORCE OPTION JUMP WITH A FAKE:
RTS
(IN THIS CASE, SELECTION "B" JUMPS TO THE OPTION STARTING AT $293C.)

There are several distinct parts to a good and flexible option picker. First, you normally will want the same response for a capital letter as for a small one, say for "A" and "a." If you do, you will need *case changer* code. If you are allowing for meaningful, rather than ordered, inputs, then you will also need an *option filter* that converts the selections into a binary file access number.

Then there are possible errors. Sometimes a few *legal* and *expected* responses may all want to go to the *same* option. We can call that jump an *inclusive trap.* The simplest way to handle inclusive traps is just to repeat the same address in the address file as often as needed. There are ways, of course, to save a byte or two on this, but you end up with custom code if you try this. We will use an error message of "PLEASE TRY ANOTHER LETTER" to show this when it happens in the upcoming GILA demo.

Other times, the input will not match any legal selection. What you have to do here is go get another input since the one you have is no good. You can call this an *exclusive trap.* Exclusive traps should go and try and get another response.

You must always inform the user that you don't like his invalid selection. The trick here is to do it as subtly and gently as possible. More often than not, a brief screen flash or a single speaker click is all you will need. We will use a message of "THAT'S NO LETTER, YOU TURKEY!" in the demo. Naturally, such harshness must be used with discretion in commercial programs.

Should the program, rather than the user, be making the option selection, you will end up in deep trouble if the computer decides to do something it is not set up to do. In *Zork,* machine errors are trapped with a "ZORK INTERNAL ERROR" message. Chances are overwhelming that you have not and will not get one of these *Zork* messages.

Unless you play *Zork* the way I do.

Needless to say, machine errors are never supposed to happen. When and if one does, though, be sure to inform the user that he has just been done in through no fault of his own. It may be a good idea to encourage the user to "close the loop" and contact you personally when this happens.

Not *if,* but *when.*

A final part of the option picker has to actually do a jump or a gosub to the selected option. You have many choices here. Building the jumps into the code as we did above is obviously bad, since the code is no longer general. You can also self-modify your code by having a JMP command whose address you pre-change to the address you want to jump to. This is risky and does not work in ROM, but is cute and compact. You can also force a JSR the same way.

The JMP indirect command is another possibility. Here, you put your address somewhere on page zero, say $06 low and $07 high. Then a JMP ($06) does an indirect jump to your intended address. For a forced subroutine, just do a JSR to an indirect JMP.

But, remember that the original 6502 JMP indirect has a bug in it that prevents you from using it properly on either of the top two bytes of any page. If you relocate your code, or do not watch very carefully where your JMP indirects are, this bug may bomb your code. Page zero real estate is valuable enough that you should go out of your way to avoid using it whenever there are reasonable alternatives.

By the way, certain copy protection fanatics intentionally put their JMP indirects in the "wrong" locations, hoping you miss the turn. The jumped-to locations end up on the bottom of the *same* page, rather than the expected bottom of the *next* page this way. Of course, such childish and inane stunts just add to the fun and challenge of cracking the "uncrackable." Besides, they will bomb on an Apple upgraded to a 65C02. Or on a IIc.

Har har.

Anyway, the way I like to do a jump to an option is with a scheme called the *forced subroutine return* method. This method is used in the Apple system monitor, so it is not new. But it is super powerful and elegant.

Remember that a subroutine return or RTS checks into the stack and gets the top stack location. It uses this location for the position on the page it is to return to. Then, it goes one deeper into the stack to get the page location. Given the position and the page, the RTS then jumps to this location *plus one.*

Normally, of course, the RTS returns to the code that called it. Now to get sneaky. Take the page address of your option and shove it on the stack with a PHA. Then, take the option position address *minus one* and shove it on the stack with a second PHA. Now, RTS. What happens?

You "return" to the address of your selected option!

Note that two pushes (by you) and two pulls (by the RTS) leave the stack the way it was before you started. So you are still in the same "level" of your code both before and after you force the fake subroutine return. Note also that no page zero locations are committed.

For an earlier and different example of using forced subroutine returns, check back into IMPRINT of Ripoff Module 2.

Let's sum up the parts of our option picker . . .

CASE CHANGER—

Code that forces lowercase letters into their uppercase equivalents.

OPTION FILTER—

Code that finds a match between user inputs and a binary value.

INCLUSIVE TRAP—

Several user selections that all divert to the same option.

EXCLUSIVE TRAP—

Code that finds "illegal" user inputs and suitably handles this type of error.

FORCED SUBROUTINE RETURN—

A JMP indirect that is faked by pushing an address pair onto the stack and then doing an RTS.

Summing up, while there are lots of ways to pick options, we will use a general and powerful file based method that is easy to use and easy to change. It is best suited for six or more unordered choices. The code is very efficient when many different selections are made.

To pick an option, you first change the case of lowercase letters, so that either a capital "A" or a lowercase "a" gets the same response. Then you look for a match in a character file. Finding the match generates a binary number useful as an address pointer.

Should a match be found, the binary number is doubled and used to access an address pair in an address file. This address pair is forced onto the stack and is then followed by an RTS, doing a jump to the selected option.

Two crucial reminders . . .

When "force feeding" a stack—

ALWAYS push the page address on first, followed by the position address.

ALWAYS use an address ONE LESS than your intended return point.

The sneaky way to automatically remove one from any address is to let your assembler's operand arithmetic handle the chore for you. Thus, instead of a label of TASKA, use TASKA−1 when defining addresses in your address file. The DW command is one good way to handle 2-byte pairs. DW automatically rearranges these pairs into their "position-page" format for you.

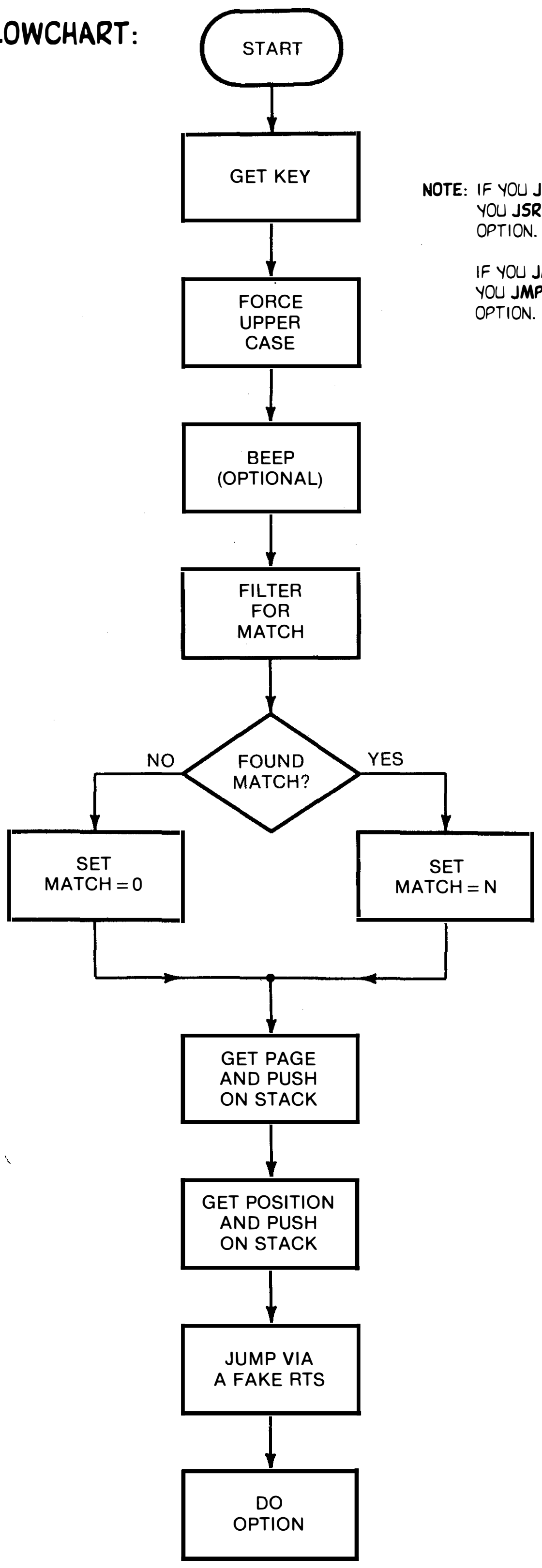
Don't forget these two crucial details: The page goes on the stack first, and RTS ends up one beyond the stack address.

Several matches can point to the same address pair by repeating the address pair when and where needed in the address file. We have seen how this is called an inclusive trap.

Should no match be found, an exclusive trap tries for a new input or generates an error message, informing the user as this happens.

The option picking subroutine is called OPICK. Its flowchart looks like this . . .

OPICK FLOWCHART:



NOTE: IF YOU **JSR** TO OPICK, YOU **JSR** TO YOUR OPTION.

IF YOU **JMP** TO OPICK, YOU **JMP** TO YOUR OPTION.

Some parts of OPICK might not be needed for all uses. For instance, you can delete getting a key if the machine itself is to provide the option selection.

Delete the case changer if you want something different to happen for a capital letter than for a lowercase one. Sometimes, your options will not even be in ASCII. They might be a binary selection. If so, lowercase is meaningless.

The option filter can be deleted if you are certain your option selections are always ordered binary numbers. This is OK for internal use, but, as we've seen, is a poor and unfriendly choice where users are involved.

And, don't use such heavy code for trivial choices. A simple (Y/N) checker can be done much faster with many fewer bytes. For over six choices, the option filter is the better way to go. The more the choices and the more wildly they are arranged, the better the method gets.

The FIXCASE case changer works by testing for a lowercase ASCII letter. If it gets one of these, then $20 is subtracted as needed to get uppercase. For instance, a lowercase "a" is an ASCII $E1. Subtract $20 to get $C1, the ASCII uppercase "A." It pays to test for "z" as well as "a" so that any punctuation above ASCII $FA does not get changed. We've shown high ASCII here, as you get off the Apple keyboard *before* resetting the keystroke.

Any match character can go in any order, except that the position of the address in the address table must be exactly *twice* the position of the match character in the match file. All this says is that the match must line up with where you want the match to go to. The doubling is needed for the 2-byte absolute address *pairs* and is handled with an ASL multiplier.

The matches in the match file can go in any order. The obvious and cleanest arrangement is to put the selections in logical user input order. Another way is to put the addresses in the order they appear in the program. Still another way is to put the often used matches first, in an attempt to gain a slight speedup. Just be sure that the match and the match address are aligned to each other.

We have put the match values in alphabetical order. Once again, though, you can put any mix of numbers, letters, and control characters in any order, skipping around anywhere you like.

As we have seen, the cleanest way to handle inclusive traps is to repeat the address pair as often as needed in your address file.

We used an inverse title for this demo, like we did with IMPRINT and FLPRINT. These titles are quick and dirty to do, but they are usually far too garish to use in a commercial program. A single inverse line cuts the tops of uppercase letters and random tops and bottoms of lowercase. The obvious cure for this of using three inverse lines to form a box usually is too "loud" for the rest of the screen.

So, do as I say, not as I do . . .

AVOID using inverse text headers and titles on commercial programs.

These are too garish and imperfect to give you acceptable results.

You have a choice of using your options as subroutines or else as same-level jumps. If, as we did in GILA, you JSR to OPICK and then force-return to your option, an RTS at the end of the option returns you to the code that is calling OPICK.

On the other hand, you could JMP to OPICK and then force-return to your option. Here, a JMP at the end of the option is needed to return you to a calling code.

In one method, the options are subroutines. In the other, they are at the same level as the code that calls OPICK.

To adapt OPICK to your own needs, just change the MAXMATCH number to equal the total number of options, change the MATCHFL file to hold the characters you are matching against, and change the JMPFLE to hold the addresses you want to jump to. A reassembly, of course, will be needed. As usual, labels that name each option you are to jump to greatly simplify and automate the process of building this file. Makes it fun even.

Time for . . .

A Demo

Normally, your option picker will jump to lots of wildly different types of code in your main program. To keep DEMO6 simple, we will still jump to lots of different points in the main program, but the action at each option point will be rather simple and sort of redundant. Now, there probably are better ways to write a program that does what this demo does, but, remember that we are trying to show the *method* of using an option picker to go many places in a larger program.

DEMO6 is our demo, and what it does is generate the name of a town in exchange for a user input that matches the first letter of that town. Once you have recovered from the initial excitement of such a stupendous program, look carefully to see how the options each go to a selected code module, and how the inclusive and exclusive traps are working. Note how both control commands and letter inputs are handled. See how the ESC key exits the program for you.

Incidentally, we've used our own key getter, rather than GETKEY. It's tricky to handle escape commands with GETKEY, and GETKEY gives slightly different results on a II versus a IIe. The prompt gets entered by using IMPRINT to print a prompt, followed by a backspace, followed by the double zero exit.

DEMO6 has borrowed the IMPRINT code from Ripoff Module 2, so be sure that either this code, a copy of it, or else a copy of THE WHOLE BALL OF WAX is present in the machine.

MIND BENDERS

—Rework the demo to use your own towns in your own local area.

—Change the demo to other topics, such as autos, aircraft, animals, vegetables, or nurflongs.

—What other uses are there for the forced subroutine method?

—How long does the option picker take to process an option?

—Show how to use your options as same level code, rather than as subroutines.

—What other user prompting can be used in place of the time delay?

—Explain away those two PLAs in the exit code. Why are they used?

—Try to BRUN OPTION PICKER directly from your disk, and [ESC] will not exit you from your program. Why?

—Display a different LORES or HIRES picture for each selection, along with suitable sound.

PROGRAM RM-6
OPTION PICKER

```
----- NEXT OBJECT FILE NAME IS OPTION PICKER
6C00:             ORG  $6C00     ; PUT MODULE #6 AT $6C00


6C00:        ;   *****************************************
6C00:        ;   *                                       *
6C00:        ;   *          -< OPTION PICKER >-          *
6C00:        ;   *                                       *
6C00:        ;   *   ( JUMPING SIX WAYS FROM SUNDAY )    *
6C00:        ;   *                                       *
6C00:        ;   *      VERSION 1.0  ($6C00-$6EDD)       *
6C00:        ;   *                                       *
6C00:        ;   *               5-24-83                 *
6C00:        ;   *.......................................*
6C00:        ;   *                                       *
6C00:        ;   *          COPYRIGHT C 1983 BY          *
6C00:        ;   *                                       *
6C00:        ;   *     DON LANCASTER AND SYNERGETICS     *
6C00:        ;   *     BOX 1300, THATCHER AZ., 85552     *
6C00:        ;   *                                       *
6C00:        ;   *     ALL COMMERCIAL RIGHTS RESERVED    *
6C00:        ;   *                                       *
6C00:        ;   *****************************************


6C00:        ;          *** WHAT IT DOES ***

6C00:        ;   THIS MODULE SHOWS YOU HOW TO JUMP TO ONE OF MANY
6C00:        ;   POSSIBLE POINTS TO CONTINUE RUNNING A PROGRAM.
6C00:        ;


6C00:        ;          *** HOW TO USE IT ***

6C00:        ;   TO USE THE OPTION PICKER:
6C00:        ;
6C00:        ;      REASSEMBLE WITH YOUR NUMBER OF MATCHES IN MATCHN,
6C00:        ;      YOUR MATCHES IN MATCHFL AND YOUR JUMP
6C00:        ;      VECTORS MINUS ONE IN JMPFL.  THEN JSR
6C00:        ;      OPICK AT $6E45 (28229).


6C00:        ;   TO RUN THE GILA TOWNS DEMO:
6C00:        ;
6C00:        ;      JSR GILA AT $6C00 OR CALL 27648.
```

PROGRAM RM-6, CONT'D . . .

```
6C00:          ;            *** GOTCHAS ***

6C00:          ;   THE IMPRINT SUBROUTINE MUST BE PRESENT IN THE
6C00:          ;   MACHINE.  PRELOAD "IMPRINT" OR "THE WHOLE BALL
6C00:          ;   OF WAX" TO DO THIS.
6C00:          ;
6C00:          ;   JUMP VECTORS MUST BE IN THE USUAL "POSITION-
6C00:          ;   PAGE" ORDER.  THE ORDER IN MATCHFL MUST EQUAL
6C00:          ;   THE ORDER IN TASKFL.
6C00:          ;   JUMP VECTORS MUST BE ONE LESS THAN
6C00:          ;   THEIR ACTUAL RETURN POINTS!


6C00:          ;           *** ENHANCEMENTS ***

6C00:          ;   MATCHED CHARACTERS CAN BE IN ANY ORDER AND MAY
6C00:          ;   INCLUDE CONTROL CHARACTERS.
6C00:          ;
6C00:          ;   INCLUSIVE TRAPS ARE DONE BY REPEATING THE TASKFL
6C00:          ;   JUMP VECTORS AS OFTEN AS NEEDED.
6C00:          ;


6C00:          ;           *** RANDOM COMMENTS ***

6C00:          ;   THERE ARE CERTAINLY BETTER WAYS TO HANDLE THE
6C00:          ;   GILA TOWNS DEMO THAN THIS.  IN REAL LIFE, EACH
6C00:          ;   "TOWN" REPRESENTS A DIFFERENT AND UNIQUE HIGH
6C00:          ;   LEVEL PROGRAMMING TASK.
6C00:          ;
6C00:          ;   THE DEMO ALSO SHOWS HOW TO CHANGE THE SCROLLING
6C00:          ;   TEXT WINDOW UNDER PROGRAM CONTROL.
```

PROGRAM RM-6, CONT'D . . .

```
6C00:          ;               *** HOOKS ***


FC58:          HOME     EQU   $FC58      ; CLEAR TEXT SCREEN AND HOME CURSOR
FB2F:          INIT     EQU   $FB2F      ; INITIALIZE TEXT SCREEN
C000:          IOADR    EQU   $C000      ; KEYBOARD INPUT LOCATION
C010:          KBDSTRB  EQU   $C010      ; KEYBOARD STROBE RESET
FE80:          SETINV   EQU   $FE80      ; SET INVERSE SCREEN
FE84:          SETNORM  EQU   $FE84      ; SET NORMAL SCREEN
C030:          SPKR     EQU   $C030      ; SPEAKER CLICK OUTPUT
FCA8:          WAIT     EQU   $FCA8      ; MONITOR TIME DELAY

666B:          IMPRINT  EQU   $666B      ; LINK TO IMPRINT SUBROUTINE


0020:          WNDLFT   EQU   $20        ; LEFT SIDE OF SCROLL WINDOW
0021:          WNDWTH   EQU   $21        ; WIDTH OF SCROLL WINDOW
0022:          WNDTOP   EQU   $22        ; TOP OF SCROLL WINDOW
0023:          WNDBTM   EQU   $23        ; BOTTOM OF SCROLL WINDOW
0024:          CH       EQU   $24        ; CURSOR HORIZONTAL POSITION
0025:          CV       EQU   $25        ; CURSOR VERTICAL POSITION
0033:          PROMPT   EQU   $33        ; PROMPT SYMBOL


6C00:          ;        *** TEXTFILE COMMANDS ***


0088:          B        EQU   $88        ; BACKSPACE
008D:          C        EQU   $8D        ; CARRIAGE RETURN
0084:          D        EQU   $84        ; DOS ATTENTION
009B:          E        EQU   $9B        ; ESCAPE
008A:          L        EQU   $8A        ; LINEFEED
0060:          P        EQU   $60        ; FLASHING PROMPT
0000:          X        EQU   $00        ; END OF MESSAGE
```

PROGRAM RM-6, CONT'D . . .

```
6C00:              ;        *** GILA TOWNS DEMO ***
6C00:              ;
6C00:              ;   THIS PROGRAM EXERCISES THE OPTION
6C00:              ;   PICKER SUBROUTINE OPICK.
6C00:              ;
6C00:              ;   EACH "TOWN" REPRESENTS A DIFFERENT
6C00:              ;   HIGH LEVEL PROGRAM TASK.


6C00:20 2F FB      GILA     JSR  INIT      ; SET UP TEXT SCREEN
6C03:20 58 FC               JSR  HOME      ; CLEAR SCREEN AND HOME CURSOR
6C06:A9 07                  LDA  #07       ; TAB 7 TO RIGHT
6C08:85 24                  STA  CH        ;
6C0A:20 80 FE               JSR  SETINV    ; INVERSE TITLE
6C0D:20 6B 66               JSR  IMPRINT   ; PUT DOWN TITLE
6C10:8A 8A 8A               DFB             L,L,L
6C13:CF D0 D4               ASC             "OPTION PICKER DEMO"
6C16:C9 CF CE
6C19:A0 D0 C9
6C1C:C3 CB C5
6C1F:D2 A0 C4
6C22:C5 CD CF
6C25:8D 8D 8D               DFB             C,C,C,C,X
6C28:8D 00

6C2A:20 84 FE               JSR  SETNORM   ; BACK TO NORMAL TEXT
6C2D:20 6B 66               JSR  IMPRINT   ; PUT DOWN INSTRUCTIONS
6C30:D4 D9 D0               ASC             "TYPE THE FIRST LETTER TO GET THE"
6C33:C5 A0 D4
6C36:C8 C5 A0
6C39:C6 C9 D2
6C3C:D3 D4 A0
6C3F:CC C5 D4
6C42:D4 C5 D2
6C45:A0 D4 CF
6C48:A0 C7 C5
6C4B:D4 A0 D4
6C4E:C8 C5
6C50:8D                     DFB             C
6C51:C6 D5 CC               ASC             "FULL NAME OF A GILA VALLEY TOWN:"
6C54:CC A0 CE
6C57:C1 CD C5
6C5A:A0 CF C6
6C5D:A0 C1 A0
6C60:C7 C9 CC
6C63:C1 A0 D6
6C66:C1 CC CC
6C69:C5 D9 A0
6C6C:D4 CF D7
6C6F:CE BA
```

PROGRAM RM-6, CONT'D . . .

```
6C71:8D 8D 8D               DFB            C,C,C,C
6C74:8D
6C75:A0 A0 A0               ASC            "         ---> "
6C78:A0 A0 A0
6C7B:A0 A0 AD
6C7E:AD AD BE
6C81:A0
6C82:8D 8D 8D               DFB            C,C,C,C,C,C
6C85:8D 8D 8D
6C88:A0 A0 A0               ASC            /      (USE "ESC" TO EXIT)/
6C8B:A0 A0 A0
6C8E:A8 D5 D3
6C91:C5 A0 A2
6C94:C5 D3 C3
6C97:A2 A0 D4
6C9A:CF A0 C5
6C9D:D8 C9 D4
6CA0:A9
6CA1:00                     DFB             X

6CA2:A9 0D                  LDA  #$0D      ; SET TIGHT WINDOW
6CA4:85 20                  STA  WNDLFT    ;
6CA6:A9 15                  LDA  #$15      ;
6CA8:85 21                  STA  WNDWTH    ;
6CAA:A9 0C                  LDA  #$0C      ;
6CAC:85 22                  STA  WNDTOP    ;
6CAE:A9 0F                  LDA  #$0F      ;
6CB0:85 23                  STA  WNDBTM    ;
6CB2:20 58 FC               JSR  HOME      ; GET IN WINDOW
6CB5:A9 60                  LDA  #P        ; CHANGE PROMPT
6CB7:85 33                  STA  PROMPT    ;

6CB9:20 6B 66      DOOPT    JSR  IMPRINT   ; ADD WINKING CURSOR
6CBC:60 88 00               DFB             P,B,X
6CBF:20 44 6E               JSR  OPICK     ; GET AND DO OPTIONS AS SUBS

6CC2:A2 0D         CONT6    LDX  #13       ; MOST OPTIONS RETURN TO HERE
6CC4:20 A8 FC      STALL6   JSR  WAIT      ; STALL FOR DISPLAY TIME
6CC7:CA                     DEX            ;
6CC8:D0 FA                  BNE  STALL6    ;

6CCA:20 58 FC               JSR  HOME      ; ERASE OLD SCREEN
6CCD:A2 28                  LDX  #$28      ;
6CCF:20 7B 6E               JSR  QUIP      ; BLORK
6CD2:4C B9 6C               JMP  DOOPT     ; AND REPEAT
```

PROGRAM RM-6, CONT'D . . .

```
6CD5:               ;        *** THE ACTUAL TASKS ***

6CD5:20 6B 66      TASKA    JSR   IMPRINT;  TASK A
6CD8:C1 D2 D4               ASC              "ARTESIA"
6CDB:C5 D3 C9
6CDE:C1
6CDF:00                     DFB              X
6CE0:60                     RTS             ;

6CE1:20 6B 66      TASKB    JSR   IMPRINT;  TASK B
6CE4:C2 CF CE               ASC              "BONITA"
6CE7:C9 D4 C1
6CEA:00                     DFB              X
6CEB:60                     RTS             ;

6CEC:20 6B 66      TASKC    JSR   IMPRINT;  TASK C
6CEF:C3 CC C9               ASC              "CLIFTON"
6CF2:C6 D4 CF
6CF5:CE
6CF6:00                     DFB              X
6CF7:60                     RTS             ;

6CF8:20 6B 66      TASKD    JSR   IMPRINT;  TASK D
6CFB:C4 D5 CE               ASC              "DUNCAN"
6CFE:C3 C1 CE
6D01:00                     DFB              X
6D02:60                     RTS             ;

6D03:20 6B 66      TASKE    JSR   IMPRINT;  TASK E
6D06:C5 C4 C5               ASC              "EDEN"
6D09:CE
6D0A:00                     DFB              X
6D0B:60                     RTS             ;

6D0C:20 6B 66      TASKF    JSR   IMPRINT;  TASK F
6D0F:C6 D2 C1               ASC              "FRANKLIN"
6D12:CE CB CC
6D15:C9 CE
6D17:00                     DFB              X
6D18:60                     RTS             ;
```

PROGRAM RM-6, CONT'D . . .

```
6D19:20 6B 66   200 TASKG    JSR  IMPRINT;  TASK G
6D1C:C7 D5 D4   201          ASC             "GUTHRIE"
6D1F:C8 D2 C9
6D22:C5
6D23:00         202          DFB             X
6D24:60         203          RTS            ;

6D25:20 6B 66   205 TASKH    JSR  IMPRINT;  TASK H
6D28:C8 C5 CC   206          ASC             "HELIOGRAPH"
6D2B:C9 CF C7
6D2E:D2 C1 D0
6D31:C8
6D32:00         207          DFB             X
6D33:60         208          RTS            ;

6D34:20 6B 66   210 TASKI    JSR  IMPRINT;  TASK I
6D37:C9 CE C4   211          ASC             "INDIAN SPRINGS"
6D3A:C9 C1 CE
6D3D:A0 D3 D0
6D40:D2 C9 CE
6D43:C7 D3
6D45:00         212          DFB             X
6D46:60         213          RTS            ;

6D47:20 6B 66   215 TASKJ    JSR  IMPRINT;  TASK J
6D4A:CA C1 C3   216          ASC             "JACKSON ESTATES"
6D4D:CB D3 CF
6D50:CE A0 C5
6D53:D3 D4 C1
6D56:D4 C5 D3
6D59:00         217          DFB             X
6D5A:60         218          RTS            ;

6D5B:20 6B 66   220 TASKK    JSR  IMPRINT;  TASK K
6D5E:CB CC CF   221          ASC             "KLONDYKE"
6D61:CE C4 D9
6D64:CB C5
6D66:00         222          DFB             X
6D67:60         223          RTS            ;

6D68:20 6B 66   225 TASKL    JSR  IMPRINT;  TASK L
6D6B:CC C9 D4   226          ASC             "LITTLE TULSA"
6D6E:D4 CC C5
6D71:A0 D4 D5
6D74:CC D3 C1
6D77:00         227          DFB             X
6D78:60         228          RTS            ;
```

PROGRAM RM-6, CONT'D . . .

```
6D79:20 6B 66  TASKM   JSR  IMPRINT;  TASK M
6D7C:CD CF D2          ASC             "MORENCI"
6D7F:C5 CE C3
6D82:C9
6D83:00                DFB             X
6D84:60                RTS            ;

6D85:20 6B 66  TASKN   JSR  IMPRINT;  TASK N
6D88:CE C1 C3          ASC             "NACHES"
6D8B:C8 C5 D3
6D8E:00                DFB             X
6D8F:60                RTS            ;

6D90:          ;       TASK O DEFAULTS TO INCTRAP

6D90:20 6B 66  TASKP   JSR  IMPRINT;  TASK P
6D93:D0 C9 CD          ASC             "PIMA"
6D96:C1
6D97:00                DFB             X
6D98:60                RTS            ;

6D99:          ;       TASK Q DEFAULTS TO INCTRAP

6D99:20 6B 66  TASKR   JSR  IMPRINT;  TASK R
6D9C:D2 CF D0          ASC             "ROPER LAKE"
6D9F:C5 D2 A0
6DA2:CC C1 CB
6DA5:C5
6DA6:00                DFB             X
6DA7:60                RTS            ;

6DA8:20 6B 66  TASKS   JSR  IMPRINT;  TASK S
6DAB:D3 C1 C6          ASC             "SAFFORD"
6DAE:C6 CF D2
6DB1:C4
6DB2:00                DFB             X
6DB3:60                RTS            ;

6DB4:20 6B 66  TASKT   JSR  IMPRINT;  TASK T
6DB7:D4 C8 C1          ASC             "THATCHER"
6DBA:D4 C3 C8
6DBD:C5 D2
6DBF:00                DFB             X
6DC0:60                RTS            ;

6DC1:          ;       TASK U DEFAULTS TO INCTRAP
```

PROGRAM RM-6, CONT'D . . .

```
6DC1:20 6B 66  268 TASKV   JSR  IMPRINT;  TASK V
6DC4:D6 C9 D2  269         ASC              "VIRDEN"
6DC7:C4 C5 CE
6DCA:00        270         DFB              X
6DCB:60        271         RTS            ;

6DCC:20 6B 66  273 TASKW   JSR  IMPRINT;  TASK W
6DCF:D7 C8 C9  274         ASC              "WHITLOCK CIENEGA"
6DD2:D4 CC CF
6DD5:C3 CB A0
6DD8:C3 C9 C5
6DDB:CE C5 C7
6DDE:C1
6DDF:00        275         DFB              X
6DE0:60        276         RTS            ;

6DE1:          278 ;       TASK X DEFAULTS TO INCTRAP

6DE1:20 6B 66  280 TASKY   JSR  IMPRINT;  TASK Y
6DE4:D9 CF D2  281         ASC              "YORK VALLEY"
6DE7:CB A0 D6
6DEA:C1 CC CC
6DED:C5 D9
6DEF:00        282         DFB              X
6DF0:60        283         RTS            ;

6DF1:          285 ;       TASK Z DEFAULTS TO INCTRAP
```

PROGRAM RM-6, CONT'D . . .

```
6DF1:20 6B 66   INCTRAP JSR  IMPRINT;  INCLUSIVE DEFAULT TRAP
6DF4:D3 CF D2           ASC             "SORRY, PLEASE TRY"
6DF7:D2 D9 AC
6DFA:A0 D0 CC
6DFD:C5 C1 D3
6E00:C5 A0 D4
6E03:D2 D9
6E05:8D                 DFB             C
6E06:D3 CF CD           ASC             "SOME OTHER LETTER"
6E09:C5 A0 CF
6E0C:D4 C8 C5
6E0F:D2 A0 CC
6E12:C5 D4 D4
6E15:C5 D2
6E17:00                 DFB             X
6E18:60                 RTS             ;

6E19:20 6B 66   ERRTRAP JSR  IMPRINT;  ILLEGAL KEY DEFAULT
6E1C:D4 C8 C1           ASC             "THATS NO LETTER
6E1F:D4 D3 A0
6E22:CE CF A0
6E25:CC C5 D4
6E28:D4 C5 D2
6E2B:8D                 DFB             C
6E2C:A0 A0 D9           ASC             "  YOU TURKEY!
6E2F:CF D5 A0
6E32:D4 D5 D2
6E35:CB C5 D9
6E38:A1
6E39:00                 DFB             X
6E3A:60                 RTS             ;

6E3B:20 2F FB   QUIT6   JSR  INIT       ; RESTORE NORMAL TEXT WINDOW
6E3E:20 58 FC           JSR  HOME       ; HOME CURSOR AND CLEAR SCREEN
6E41:68                 PLA             ; BYPASS GILA; GO STRAIGHT
6E42:68                 PLA             ; TO MONITOR OR CALLING CODE
6E43:60                 RTS             ; FOR COMPLETE EXIT
```

PROGRAM RM-6, CONT'D . . .

```
6E44:              ;        *** OPTION PICKER SUBROUTINE ***
6E44:              ;
6E44:              ;    FOR OTHER USES, THIS SUB HAS TO BE LINKED TO
6E44:              ;    YOUR OWN MATCHN MATCH NUMBER, YOUR MATCHF
6E44:              ;    CHARACTER MATCHER FILE AND YOUR JMPFLE VECTORS.
6E44:              ;


6E44:2C 10 C0   OPICK   BIT  KBDSTRB    ; LOCK OUT EARLY HITS
6E47:AD 00 C0   LOOK6   LDA  IOADR      ; GET KEY.  CAN'T USE KEYIN
6E4A:10 FB              BPL  LOOK6      ;  BECAUSE WE NEED ESC COMMAND.
6E4C:2C 10 C0           BIT  KBDSTRB    ; RESET STROBE

6E4F:2C 6F 6E           JSR  FIXCASE    ; FORCE UPPERCASE

6E52:A2 0A              LDX  #10
6E54:20 7B 6E           JSR  QUIP       ; BLORK

6E57:AE 8A 6E           LDX  MATCHN     ; GET LEGAL NUMBER OF MATCHES
6E5A:DD 8B 6E   SCAN6   CMP  MATCHFL,X  ; SEARCH FOR A MATCH
6E5D:F0 03              BEQ  GOTMTCH    ; FOUND
6E5F:CA                 DEX             ; TRY NEXT
6E60:10 F8              BPL  SCAN6;

6E62:E8         GOTMTCH INX             ; MAKES ZERO A MISS
6E63:8A                 TXA             ; GET JUMP VECTOR
6E64:0A                 ASL  A          ; DOUBLE POINTER
6E65:AA                 TAX             ;
6E66:BD A7 6E           LDA  JMPFL+1,X  ; GET PAGE ADDRESS FIRST!
6E69:48                 PHA             ;  AND FORCE ON STACK
6E6A:BD A6 6E           LDA  JMPFL,X    ; GET POSITION ADDRESS
6E6D:48                 PHA             ;   AND FORCE ON STACK
6E6E:60                 RTS             ; JUMP VIA FORCED SUBROUTINE RETURN
```

PROGRAM RM-6, CONT'D . . .

```
6E6F:           ;       *** CASE FIXER SUBROUTINE ***

6E6F:           ;   TESTS THE ACCUMULATOR FOR A LOWERCASE
6E6F:           ;   CHARACTER.  IF PRESENT, FORCES UPPERCASE
6E6F:           ;   BY ADDING $20.  USES HIGH ASCII.

6E6F:C9 E1      FIXCASE CMP  #$E1      ; IF "a" OR MORE
6E71:90 07              BCC  NOFIX6    ;
6E73:C9 FB              CMP  #$FB      ;  AND IF "z" OR LESS
6E75:B0 03              BCS  NOFIX6    ;
6E77:38                 SEC            ;  THEN SUBTRACT $20 TO
6E78:E9 20              SBC  #$20      ;  FORCE UPPER CASE
6E7A:60         NOFIX6  RTS            ;  AND RETURN



6E7B:           ;       *** QUIP SUBROUTINE ***
6E7B:           ;
6E7B:           ;    MAKES NOISE. X SETS THE PITCH.  THE
6E7B:           ;     PITCH IS PROPORTIONAL TO THE DURATION.
6E7B:           ;     WHICH IS OK FOR THIS SIMPLE USE BUT
6E7B:           ;     SHOULD BE AVOIDED MOST EVERYWHERE ELSE.

6E7B:48         QUIP    PHA            ; SAVE ACCUMULATOR
6E7C:A0 3C              LDY  #60       ; NUMBER OF CYCLES
6E7E:8A         NXT6    TXA            ; PITCH
6E7F:2C 30 C0           BIT  SPKR      ; WHAP SPEAKER
6E82:20 A8 FC           JSR  WAIT      ;
6E85:88                 DEY            ; NEXT CYCLE
6E86:D0 F6              BNE  NXT6      ;
6E88:68                 PLA            ; RESTORE ACCUMULATOR
6E89:60                 RTS            ; AND EXIT
```

PROGRAM RM-6, CONT'D . . .

```
6E8A:            ;        *** OPTION PICKER FILES ***


6E8A:            ;       MATCHN HOLDS THE NUMBER OF MATCHES.
6E8A:            ;       MATCHFL HOLDS THE LEGAL CHARACTERS.
6E8A:            ;       JUMPFL HOLDS THE JUMP VECTORS.
6E8A:            ;
6E8A:            ;       NOTE THAT ANY NUMBER OF CHARACTERS
6E8A:            ;       AND CONTROL COMMANDS MAY BE USED
6E8A:            ;       IN ANY ORDER, BUT THAT EACH MUST
6E8A:            ;       POSITION MATCH ITS JUMPFL VECTOR.


6E8A:1B          MATCHN  DFB  27          ; NUMBER OF LEGAL MATCHES GOES HERE


6E8B:9B          MATCHFL DFB  E           ; FOR ESCAPE

6E8C:C1 C2 C3            ASC               "ABCDEFGHIJKLM"
6E8F:C4 C5 C6
6E92:C7 C8 C9
6E95:CA CB CC
6E98:CD

6E99:CE CF D0            ASC               "NOPQRSTUVWXYZ"
6E9C:D1 D2 D3
6E9F:D4 D5 D6
6EA2:D7 D8 D9
6EA5:DA


6EA6:18 6E       JMPFL   DW   ERRTRAP-1 ; NOT A LEGAL KEY
6EA8:3A 6E               DW   QUIT6-1   ; EXIT ON ESCAPE
6EAA:D4 6C               DW   TASKA-1   ; DO LETTERED TASK
6EAC:E0 6C               DW   TASKB-1   ;
6EAE:EB 6C               DW   TASKC-1   ;
6EB0:F7 6C               DW   TASKD-1   ;
6EB2:02 6D               DW   TASKE-1   ;
6EB4:0B 6D               DW   TASKF-1   ;
6EB6:18 6D               DW   TASKG-1   ;
6EB8:24 6D               DW   TASKH-1   ;
6EBA:33 6D               DW   TASKI-1   ;
6EBC:46 6D               DW   TASKJ-1   ;
6EBE:5A 6D               DW   TASKK-1   ;
6EC0:67 6D               DW   TASKL-1   ;
6EC2:78 6D               DW   TASKM-1   ;
6EC4:84 6D               DW   TASKN-1   ;
```

PROGRAM RM-6, CONT'D . . .

```
6EC6:F0 6D          DW   INCTRAP-1 ; LEGAL BUT NO TOWN
6EC8:8F 6D          DW   TASKP-1   ;
6ECA:F0 6D          DW   INCTRAP-1 ; LEGAL BUT NO TOWN
6ECC:98 6D          DW   TASKR-1   ;
6ECE:A7 6D          DW   TASKS-1   ;
6ED0:B3 6D          DW   TASKT-1   ;
6ED2:F0 6D          DW   INCTRAP-1 ; LEGAL BUT NO TOWN
6ED4:C0 6D          DW   TASKV-1   ;
6ED6:CB 6D          DW   TASKW-1   ;
6ED8:F0 6D          DW   INCTRAP-1 ; LEGAL BUT NO TOWN
6EDA:E0 6D          DW   TASKY-1   ;
6EDC:F0 6D          DW   INCTRAP-1 ; LEGAL BUT NO TOWN



*** SUCCESSFUL ASSEMBLY: NO ERRORS
```

7

RANDOM NUMBERS

pseudo-random number generator is fast, flexible, and free of defects

Random numbers are essential for many computer uses, from the throw of a die, through animated game motions, to industrial simulations. How can you introduce randomness into your programs?

It turns out that there are two types of "random" numbers. A *real* random number is a number that can be one of many equally likely values. A *pseudo-random* number is the next number available in a contrived series that appears on the surface to be any one of many equally likely values . . .

RANDOM NUMBER—

A number that can assume any one of many equally likely values.

PSEUDO-RANDOM NUMBER—

The next number available in a contrived series that appears on the surface to be any one of many equally likely values.

The advantages of "real" random numbers is that they are truly unpredictable. Disadvantages of real random numbers include that they are hard or inconvenient to generate and that there is no way to get the same random sequence back over again at a later time.

There is a very simple and very useful real random number generator built into your Apple. Any time you use the monitor subroutine KEYIN, there is a 16-bit counter involving locations RNDL and RNDH that gets incremented a random number of times. The randomness comes about since there is no control over how long a user waits between keystrokes. RNDL is located at $4E and RNDH is located at $4F on page zero. The monitor routines GETLN, GETLNZ, GETLN1, RDCHAR, and RDKEY all use KEYIN, so any of these can be used to fetch a new random number . . .

To generate a real random number with your Apple, use the monitor routine KEYIN and then read the 16-bit true random result at $4E and $4F.

The result is a truly random 16-bit number every time. For a new random number, have the user make repeated use of KEYIN, such as with a "HIT ANY KEY TO CONTINUE," or even start out with the flea-bitten "HI, WHAT'S YOUR NAME?" prompt.

If you don't need the full 16 bits, just mask off those you do want. One bit gives you a yes-no decision. A pair of bits generates the random digits from 0–3 and so on. For a RND(6), do a RND(8) instead, and, if you get a 6 or 7 result, go fish again. Or, better yet, you could also write your own version of GETKEY that counts your own base six counter round and round.

Same goes for any other modulo.

By modulo, we mean . . .

MODULO—

The "N" in RND (N).

Note that RND(N) returns with one of N possible values, ranging from ZERO to ONE LESS THAN N.

Uh, better repeat that. The modulo is the total number of different random numbers you can get back. Since zero is always one of them, the range of numbers will go from zero to N−1.

You never get a value of N for RND(N).

At any rate, using RNDL and RNDH, or else your own software counter for true randomness is very simple. But, there are at least two big disadvantages.

First and worst, the user must hit a key for *every* new random number you need. This gets old fast if more than a dozen selections are involved. Sometimes you can disguise what's happening in a game where lots of keystrokes are involved, but not often.

Secondly, this is a slow process that takes many milliseconds. You can generate pseudo-random numbers hundreds or even thousands of times faster.

And, finally, there is no way to get the same random numbers back again in the same sequence, for replays, or for "noise that repeats."

So, while you have a true random number generator in your Apple and while it is very simple to access, you may not be able to do very much with it.

What About Applesoft's RND?

The advantages of pseudo-random number sequences are that they are easy to generate, and you can easily get the same short and apparently "random" sequence back as often as you like. This is handy for replaying a hand of cards, or to provide "noise that repeats" for industrial testing. You can also do this much faster than you can waiting for someone to press a key.

Applesloth has a subroutine in it that is a failed attempt at pseudo-random number generation.

By now, just about everyone knows that there is a fatal flaw in the Applesloth random number generator, that causes things to repeat in an annoying and frustratingly short way. And, no, the published fixes don't help enough to be useful. So, besides it taking forever to generate a random number, this subroutine simply does not work . . .

APPLESOFT RND AIN'T.
DON'T USE IT!

The fundamental problem is twofold. First, and more or less fixable, the Applesloth RND function does not "reseed" itself every time. The published repairs help this bunches, by using RNDL and RNDH as seeds.

Secondly, and fatally, any pseudo-random sequence generator is supposed to work by making the sequence so long that the numbers will apparently "never" repeat. For many argument values, the Applesloth RND generator does in fact generate an acceptably long sequence. But there are some exactly wrong magic values that repeat in as short as 200 or fewer values! And, as anyone who has used RND knows, these short sequences happen often enough to be a serious problem.

Actually, it is super difficult to fake generation of "random" numbers. There is level upon level of subtlety in the math involved in proving that any system for generating pseudo-random numbers is in fact able to provide truly random results.

What we should be worried about is something useful enough to appear random, even if it might eventually fail some exotic randomness test. It turns out that there is a very simple and devastatingly powerful way to test for randomness. Just put random dots on the HIRES screen. If the screen turns white, you are well on your way to having a good random number generator. If it gets lines, large patterns, or shading in it during this test, you have preferential numbers. If the screen "sticks" and never gets to all white, your sequence is too short to be useful and is repeating itself.

This simple scheme uses your eye as an optical correlator to really pull any nonrandomness right out of the woodwork.

Applesloth's RND always fails the screen test. Sometimes it fails it quickly, "sticking" after as few as 200 dots. Other times, you will get thousands of dots on the screen before the sequence repeats. The worst results are gotten by rerunning the same sequence over and over again.

Want to try it? . . .

```
3  REM
        DEMO TO SHOW WHY RND AINT

4  REM
        * RUN IT TILL IT STICKS *

5  HGR : HCOLOR = 3: REM

10 X = 280 * RND (1): Y = 192 * RND
     (1): HPLOT X,Y: GOTO 10
```

There are a few [J]'s in and amongst the code in this listing for pretty printing. Leave them off if you care to. Unless you immediately happen into the short sequences, the program may have to run a few minutes before it sticks.

Actually, to be fair, Applesloth is stuck with doing floating point random number generation, which is a far stickier problem than simply generating one number from a small integer field.

An Integer Pseudo-Random Generator

Let's instead worry about generating integer pseudo-random numbers. The method we will show you easily handles any value from RND (2) to RND (255), and is extremely fast. It passes the screen-fill test with flying colors.

First, some theory. We will use a method called the *shift register pseudo-random sequence generator* method. This one is detailed both in the *TTL Cookbook* and the *CMOS Cookbook* (Sams 21035 and 21398).

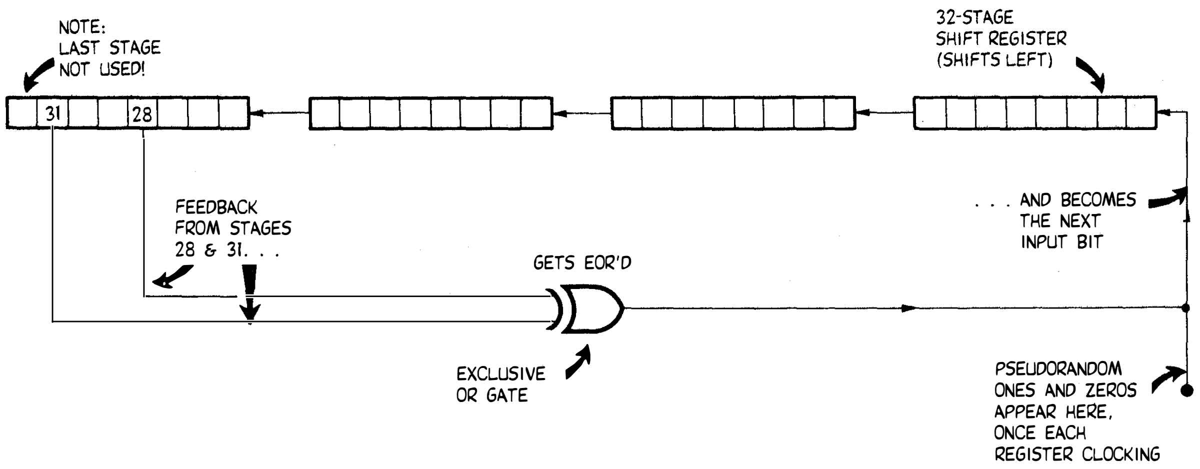
There is a hardware beastie called a shift register that can be made to behave like a counter. By taking certain high taps off the shift register and EXCLUSIVE-ORing them together and feeding these back to the input, you can generate a very long sequence.

Very handily, any tests you make on a short burst in the sequence will lead you to believe you have a true random number generator. The optimum feedback connections lead to a *maximal length* sequence, which turns out to be one less than two raised to the number of stages in use.

To get "random" numbers, you keep picking up new numbers in the sequence, or else jump to some other wildly different place in the series. To get replays or noise that repeats, you start over again at the same point in the series you did before.

We will use a 31-stage pseudo-random register since the feedback needed is simpler than that needed by a 32-stage one. The hardware we are going to synthesize with software looks like so . . .

A HARDWARE WAY TO GENERATE "RANDOM" NUMBERS:



(SEQUENCE LENGTH = 2,147,483,647)

There are 31 stages to our register. We take the output from stage 31 and EXCLUSIVE-OR it with the output from stage 28. The EOR of these taps then becomes the new value fed back to the input. These stage taps are "magic" values; anything else won't give you a super long series. We've shown this as a "shift-left" register, so we can be comparable to the replacement software we are about to use.

The sequence you get is one less than 2^31, which translates to 2,147,483,647 counts before repeating. The variable sequence length of the Applesloth code is avoided, since you have one and only one long sequence, rather than bunches, a few of which can end up short.

This shift register can be thought of as a bit pipe or stream with two billion marbles in it, half red and half white. Grab any four marbles in sequence and you have a 4-bit random number. Grab the next four and you have a new 4-bit "random" number, and so on. In this case, you can get half a billion different 4-bit random numbers in sequence before the same marbles start coming back out. And, in fact, you will get four different half billion number sequences that are predictably related but not the same, since you are one marble short at the end of the first run, and so on.

Bunches, at any rate.

There is only one little gotcha to using a generator like this. What about the missing count? It turns out that . . .

A GOTCHA—

A pseudo-random sequence generator will hang if it ever gets into the "all zeros" state.

DON'T LET THIS HAPPEN!

Now, the odds are only one in two billion of this ever happening, but you should know about it, and should prevent this hangup from ever happening. All you do is make sure there is a one somewhere in your shift register before you begin.

We will use software rather than hardware here. Set aside four bytes for the needed 31 bits. Use the EOR command for the EXCLUSIVE-OR logic, and use shift commands to move the bits from stage to stage.

Some Code

It takes more than just a pseudo-random generator to make a good random number generation system.

First, we should have some way of initializing or reseeding the PSR 4-byte shift register. We do this by grabbing two bytes from RNDL and RNDH that are truly random, and by grabbing two more bytes off the last PSR state.

Secondly, we need some way to get an old sequence back for replays and noise that repeats. To do this, we keep a copy of the old reseeding in a separate 4-byte seed register. For a new sequence, you load the PSR from the reseeder. For a ''used'' or repeat sequence, you reload the PSR from the seed register.

Thirdly, we need some way to deal with nonbinary numbers. A RND (32) is fairly trivial, since 32 is a binary number, and we expect a result anywhere between 0 and 31. To do this, just whump the PSR register five times, once for each bit, and read the bits with a $1F mask (that's 0001 1111 in binary) to get your result. For different binary lengths, use different mask lengths. The magic mask values are $01, $03, $07, $0F, $1F, $3F, $7F, and $FF.

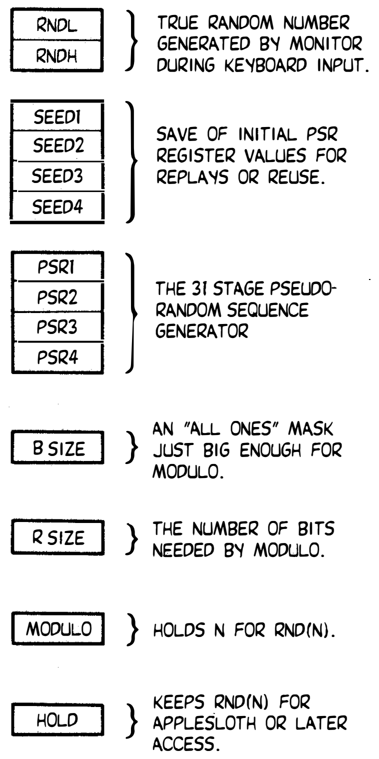
But what about a RND (26)?

Here we expect a result between decimal 0 and 25, or between hex $00 and $19. What you do is use a mask to grab more than enough bits off the PSR, and then compare the result. If the result is in range, use it. If not, go fish. Repeat the process as often as you have to.

Which I'm not very proud of, but it works. For nonbinary values, there will be some chance of having to repeat the process. This chance is always less than 50 percent worst case, and typically, is much better. So, you will still get a fast result on any RND choice although binary values will be the fastest.

Since there are lots of pieces to this randomizer, let's first look at our working stashes to see what they tell us . . .

STASHES USED BY RANDOM:


RNDL
RNDH
TRUE RANDOM NUMBER GENERATED BY MONITOR DURING KEYBOARD INPUT.
SEED1
SEED2
SEED3
SEED4
SAVE OF INITIAL PSR REGISTER VALUES FOR REPLAYS OR REUSE.
PSR1
PSR2
PSR3
PSR4
THE 31 STAGE PSEUDO-RANDOM SEQUENCE GENERATOR
BSIZE
AN "ALL ONES" MASK JUST BIG ENOUGH FOR MODULO.
RSIZE
THE NUMBER OF BITS NEEDED BY MODULO.
MODULO
HOLDS N FOR RND(N).
HOLD
KEEPS RND(N) FOR APPLESLOTH OR LATER ACCESS.

There are fourteen stash values involved.

The actual PSR generator is labeled PSR1 through PSR4. We input to the low bit of PSR1 and feedback from bits 28 and 31 that are stashed in PSR4.

There are four seed bytes used to hold the previous starting point for the PSR sequence. These are called SEED1 through SEED4. These locations are seeded from the monitor's RNDL and RNDH.

The location called MODULO holds your RND argument. For instance, on a single die, use a MODULO value of six. In return, you will get one of the six possible equiprobable states from zero to five back. MODULO must be set on first use, but if you want the same random range over and over again, you do not have to change it.

The locations called RSIZE and BSIZE take some explanation, since they are the key to generating nonbinary random values. BSIZE is a mask of enough ones to equal one less than the next higher binary power of the number you are after. That's one of those magic \$01, \$03, \$07 . . . through \$FF values. BSIZE is automatically calculated for you when and as needed. RSIZE is a save of the number of PSR advances needed to get enough bits to handle your MODULO.

For instance, on a die, BSIZE will be a \$07, or binary %0000 0111, while RSIZE will be three. Why? Because it will take three bits to generate one of the numbers from zero to five. Should we overdo ourselves and get a six or seven result, we go fish and try again. The odds of hitting a legal value in this case are 3/4 of the time on the first try, and 15/16 of the time by the second try.

The reason you want to keep BSIZE as small as possible is so your odds of a hit are high. If, instead, you tried for six values out of a possible 256, your odds on a first-try hit will be a miniscule 6/256. The reason for a separate save of RSIZE is so you do not have to recalculate BSIZE for each entry.

Which speeds things up bunches.

There are several places where our code *falls through* to another routine . . .

FALLING THROUGH—

Code that automatically goes on and does a second task.

You also have the option of doing only the second task by itself.

There are three parts to the pseudo-random generator code. These are the reseeder, the N initializer, and the actual PSR generator.

Each part is simple enough that you should be able to work up your own flowcharts.

The reseeder is used to move your position in the PSR sequence either to where you last started counting, or else to some wildly new point.

You should always JSR to this code anytime you want to start randomizing something new. If you do a JSR RESEED, you will shuffle the deck and begin at some unknown point in the PSR sequence. If you do a JSR RESET, you will reload the last seed value you used. Use RESEED for something entirely new. Use RESET to repeat the last sequence of random numbers for a replay or for noise that repeats.

Note how RESEED falls through to RESET. Note also that we make sure that PSR2 is not a zero value. If it is, we force it to one. This is one heavy way to be sure that you never hit the all ones gotcha in your PSR generator.

Every time you start up, or every time you change your modulo, you will have to activate the N initializer. Do this by a JSR to RANDOM, after putting the number of possible values you are after into MODULO. MODULO must be at least one. If it is zero, an error trap increments it.

Now, the code in the N initializer is admittedly obtuse, but this is what it does: Your modulo is scanned to generate a BSIZE mask with just enough sequential ones in it to equal or exceed your modulo value. At the same time, the number of bits involved is saved as RSIZE.

For instance, say you want RND (10). Modulo will be ten, and you expect the ten digits from zero to nine back. BSIZE will be %0000 1111, since this is the smallest mask you can have that can isolate all the digits from zero to nine. RSIZE will be four, since four random bits are needed from the PSR generator.

The N initializer falls through to the "real" PSR generator. You have to use this initializer any time you first begin or any time you change your modulo.

There are two parts to the "real" PSR generator. The first half, labeled REUSEN, gets enough bits to be equal to or more than your modulo.

To do this, REUSEN first "aligns" bit 28 to bit 31 of PSR4 using the accumulator to shift bit 28 over three places. An EOR then computes the feedback term. This particular EOR term gets shifted into the carry.

Note that there are seven worthless EOR bit results calculated at the same time. These are simply ignored. The good result ends up in the carry flag; everything else gets flushed.

Our carry flag now holds our feedback product. To shift our shift register, you move carry into the least significant bit of PSR1 and shift the rest of PSR1's bits one to the left. The high bit now goes into the carry. Now shift, in turn PSR2, PSR3, and PSR4. The net result is that you faked a pseudo-random shift register with software.

The PSR is shifted as often as you have to, once for each count of RSIZE. Three whaps for a die, four for modulo ten, and so on.

At this point, enough bits have been randomized in PSR1 to give you a totally new number equal to or larger than your modulo.

The second half of the PSR process is called RANGE. Its purpose is to see if you are within your modulo with the present random value in PSR1. If you are in range, you are finished. If not, you have to repeat the process as often as you need to for a useful result. As we've seen, the odds on a hit are always greater than 50 percent and are usually much greater.

RANGE grabs the value in PSR1 and masks it with BSIZE. This cuts the number down to size, such as to four bits for a modulo of ten. The same four bits could be used for any modulo from nine through sixteen.

The result in the accumulator is then compared to MODULO. If you are *less than* MODULO, then your value is acceptable. If you are *equal to or greater than* MODULO, then your present value is no good, and you need another trip back through the PSR.

Two minor points. Note that you do not have to go back through the N initializer for a repeat trip, since you already know and have saved BSIZE and RSIZE. This speeds things up considerably. Secondly, note that you always want less than MODULO as a result, because of a possible zero answer. To repeat, if your MODULO is six, you get any of the six different values of zero, one, two, three, four, or five. You do *not* want an answer of ''six'' for a MODULO of six, since this is the *seventh,* and not the sixth possible value.

Summing up, to generate a ''random'' number, put the range of that number into MODULO. If this is your first time through, do a JSR RESEED. If you want to repeat a previous series, do a JSR RESET. Then do a JSR RANDOM. Your result ends up in the accumulator for machine language use, and in HOLD either for high level language access or for future reference.

If you want to use the same modulo over again, do a JSR REUSEN. This is much faster.

The companion demo is called FILL. It fills the HIRES screen the same way that WHY RND AIN'T didn't. Note particularly the speed difference, how clean the process is, and how you eventually get to a totally white screen.

Even this speed test is hardly fair, since we are still using the ludicrously slow Applesloth HPLOT subroutines in the demo. You can go much faster if you add your own custom HPLOT code.

You can extend your modulo to 511 by making two trips to RANDOM. To do this, divide the *even* part of your modulo *by two.* Generate this value and double it. Then do a separate RND (2) to pick even or odd results.

MIND BENDERS

- Show how to eliminate the repeat trips for nonbinary N.
- What is the actual speed involved in generating a random number?
- How can you use a PSR generator to generate speaker noise?
- Why and how does a 23-bit PSR fail the screen-fill test?
- Can you think of any uses for shorter or longer PSR generators?

PROGRAM RM-7
RANDOM NUMBERS

```
----- NEXT OBJECT FILE NAME IS RANDOM
6F00:                  ORG  $6F00      ; PUT MODULE #7 AT $6F00


6F00:             ; *****************************************
6F00:             ; *                                       *
6F00:             ; *              -< RANDOM >-             *
6F00:             ; *                                       *
6F00:             ; *    (PSEUDORANDOM INTEGER GENERATOR)   *
6F00:             ; *                                       *
6F00:             ; *      VERSION 1.0  ($6F00-$6FB2)       *
6F00:             ; *                                       *
6F00:             ; *               1-12-83                 *
6F00:             ; *.......................................*
6F00:             ; *                                       *
6F00:             ; *          COPYRIGHT C 1983 BY          *
6F00:             ; *                                       *
6F00:             ; *     DON LANCASTER AND SYNERGETICS     *
6F00:             ; *     BOX 1300, THATCHER AZ., 85552     *
6F00:             ; *                                       *
6F00:             ; *     ALL COMMERCIAL RIGHTS RESERVED    *
6F00:             ; *                                       *
6F00:             ; *****************************************


6F00:             ;          *** WHAT IT DOES ***

6F00:             ; THIS MODULE GIVES YOU A PSEUDORANDOM INTEGER FROM A
6F00:             ; FIELD OF N.  N CAN RANGE FROM 2 TO 255.
6F00:             ;


6F00:             ;          *** HOW TO USE IT ***

6F00:             ; TO RESEED (INITIALIZE) FROM A TRUE RANDOM NUMBER,
6F00:             ; DO A JSR SEED WITH A JSR $6F2E OR A CALL 28462.
6F00:             ;
6F00:             ; TO REPEAT AN OLD PSEUDORANDOM SERIES, DO A JSR RESET
6F00:             ; BY DOING A JSR $6F44 OR A CALL 28484.
6F00:             ;
6F00:             ; TO GET A PSEUDORANDOM VALUE:
6F00:             ;
6F00:             ;   FROM MACHINE LANGUAGE, PUT N IN THE ACCUMULATOR
6F00:             ;   AND THEN JSR RANDOM AT $6F5B.  RND(N) RETURNS IN A.
6F00:             ;
6F00:             ;   FROM APPLESOFT, STORE N IN MODULO AT 28593
6F00:             ;   AND THEN CALL RANDHL AT 28504.  RND(N) ENDS
6F00:             ;   UP IN HOLD AT 28594.
```

PROGRAM RM-7, CONT'D . . .

```
6F00:             ;              *** GOTCHAS ***

6F00:             ;   HALF THE ORIGINAL RANDOM SEED COMES FROM RNDL AND
6F00:             ;   RNDH IN THE MONITOR.  THE OTHER HALF COMES FROM
6F00:             ;   THE PREVIOUS PSR SEQUENCE.
6F00:             ;   N VALUES ONE LESS THAN A BINARY POWER EXECUTE FASTEST.
6F00:             ;   APPLESOFT IS NEEDED FOR THE SCREENFILL DEMO.
6F00:             ;   THE A AND Y REGISTERS ARE USED BY THESE SUBS.


6F00:             ;             *** ENHANCEMENTS ***

6F00:             ;   THE DEMO "FILL" LETS YOU FILL THE HIRES SCREEN RANDOMLY.
6F00:             ;   RUN IT WITH A JSR $7E00 OR A CALL 332256.
6F00:             ;


6F00:             ;             *** RANDOM COMMENTS ***

6F00:             ;   VALUES OF N THAT ARE NOT EQUAL TO ONE LESS THAN A
6F00:             ;   POWER OF TWO MAY NEED REPEAT TRIPS THROUGH THE PSR
6F00:             ;   SEQUENCER.  THIS IS DONE AUTOMATICALLY.  THE PROBABILITY
6F00:             ;   OF A HIT ALWAYS EXCEEDS 50% WORST CASE PER PASS AND
6F00:             ;   IS USUALLY MUCH HIGHER.
6F00:             ;
```

PROGRAM RM-7, CONT'D . . .

```
6F00:          ;          *** HOOKS ***

F3E2:          HGR     EQU  $F3E2     ; APPLESOFT CLEAR TO HIRES ONE
F457:          HPLOT   EQU  $F457     ; APPLESOFT HIRES PLOT
C000:          IOADR   EQU  $C000     ; KEYBOARD
C010:          KBSTR   EQU  $C010     ; KEYBOARD RESET
004E:          RNDL    EQU  $4E       ; RANDOM NUMBER LOW
004F:          RNDH    EQU  $4F       ; RANDOM NUMBER HIGH
F6EC:          SETHCOL EQU  $F6EC     ; APPLESOFT HIRES COLOR SET
C050:          TEXT    EQU  $C050     ; TEXT SCREEN



6F00:          ;        *** CONSTANTS ***

0003:          COLOR   EQU  $03       ; FOR A WHITE PLOT
```

PROGRAM RM-7, CONT'D . . .

```
6F00:             ;            *** SCREENFLL DEMO ***

6F00:             ;       THIS DEMO FILLS THE HIRES SCREEN ONE RANDOM
6F00:             ;       DOT AT A TIME.
6F00:             ;
6F00:             ;
6F00:             ;
6F00:             ;

6F00:20 E2 F3     FILL    JSR  HGR        ; CLEAR HIRES SCREEN
6F03:A2 03                LDX  #COLOR     ; PICK COLOR (03=WHITE)
6F05:20 EC F6             JSR  SETHCOL    ;

6F08:20 2E 6F             JSR  RESEED     ; SEED PSR FROM RNDL,RNDH

6F0B:A9 BF        PLOTDOT LDA  #$BF       ; 191 DOTS HIGH
6F0D:8D B1 6F             STA  MODULO     ;
6F10:20 5B 6F             JSR  RANDOM     ; GET RANDOM H
6F13:48                   PHA             ; AND SAVE ON STACK
6F14:A9 FF                LDA  #$FF       ; 256 DOTS WIDE
6F16:8D B1 6F             STA  MODULO     ;
6F19:20 5B 6F             JSR  RANDOM     ; GET VERT
6F1C:A0 00                LDY  #$00       ; NO HISCREEN
6F1E:AA                   TAX             ; TRANSFER H
6F1F:68                   PLA             ; GET V
6F20:20 57 F4             JSR  HPLOT      ; PLOT DOT ON SCREEN
6F23:2C 00 C0             BIT  IOADR      ; READ KEYBOARD
6F26:30 02                BMI  EXIT7      ;
6F28:10 E1                BPL  PLOTDOT    ; CONTINUE IF NO KP
6F2A:2C 10 C0     EXIT7   BIT  KBSTR      ; RESET KEYBOARD
6F2D:60                   RTS             ; AND QUIT
```

PROGRAM RM-7, CONT'D . . .

```
6F2E:          ;           *** PSEUDORANDOM GENERATOR ***

6F2E:          ;  THE PSEUDORANDOM GENERATOR IS A REGISTER THAT IS 31
6F2E:          ;  BITS LONG.  BITS 28 AND 31 ARE EXCLUSIVE ORED TO SET
6F2E:          ;  THE NEXT MSB.  SEQUENCE LENGTH IS 2,147,483,647.
6F2E:          ;
6F2E:          ;
6F2E:          ;
6F2E:          ;


6F2E:          ;           *** THE RESEEDER ***


6F2E:A5 4E       RESEED  LDA  RNDL       ; GET RANDOM NUMBER
6F30:8D A7 6F            STA  SEED1      ; FROM MONITOR KEYBOARD RND
6F33:A5 4F               LDA  RNDH       ; AND STORE FOR PSR SEED.
6F35:8D AA 6F            STA  SEED4      ;
6F38:AD AD 6F            LDA  PSR3       ; RESEED MIDDLE FROM OLD
6F3B:8D A8 6F            STA  SEED2      ;
6F3E:AD AC 6F            LDA  PSR2       ;
6F41:8D A9 6F            STA  SEED3      ; AND FALL THRU TO RESET


6F44:A0 04       RESET   LDY  #$04       ; MOVE SEED TO PSR REGISTER
6F46:B9 A7 6F    NXT7    LDA  SEED1,Y    ;
6F49:99 AB 6F            STA  PSR1,Y     ;
6F4C:88                  DEY             ;
6F4D:D0 F7               BNE  NXT7       ;

6F4F:AD AC 6F            LDA  PSR2       ; FORCE PSR SEED TO NONZERO
6F52:D0 03               BNE  DONE7      ; BY FORCING NONZERO PSR2
6F54:EE AC 6F            INC  PSR2       ;
6F57:60          DONE7   RTS             ; AND RETURN
```

PROGRAM RM-7, CONT'D . . .

```
6F58:           ;          *** THE N INITIALIZER ***

6F58:AD B1 6F      RNDHL   LDA  MODULO      ; ENTER HERE FROM APPLESOFT
6F5B:8D B1 6F      RANDOM  STA  MODULO      ; ENTER HERE FROM MACHINE LANGUAGE
6F5E:D0 05                 BNE  BSCALC      ; N MUST NOT BE ZERO!
6F60:A9 02                 LDA  #$02        ; USE N=2 MINIMUM
6F62:8D B1 6F              STA  MODULO      ;

6F65:A9 FF         BSCALC  LDA  #$FF        ; INIT SIZE TO 255
6F67:8D AF 6F              STA  BSIZE       ; ENOUGH ONES HERE > MODULO
6F6A:A0 08                 LDY  #$08        ; FOR 8 BITS
6F6C:AD B1 6F              LDA  MODULO      ; GET MODULO AND CALCULATE
6F6F:2A            SMALLER ROL  A           ; NEXT LARGER
6F70:B0 0C                 BCS  ADVANCE     ;
6F72:4E AF 6F              LSR  BSIZE       ; DIVIDE BY TWO
6F75:88                    DEY              ; NEXT SMALLER
6F76:D0 F7                 BNE  SMALLER     ;
6F78:8C B0 6F              STY  RSIZE       ; SAVE FOR RETRY



6F7B:           ;          *** THE ACTUAL PSR GENERATOR ***

6F7B:AC B0 6F      REUSEN  LDY  RSIZE       ; RESTORE IF RETRY
6F7E:AD AE 6F      ADVANCE LDA  PSR4        ; GET HIGH PSR
6F81:0A                    ASL  A           ; ALIGN BIT 28 TO 31
6F82:0A                    ASL  A           ;
6F83:0A                    ASL  A           ;
6F84:4D AE 6F              EOR  PSR4        ; AND EXCLUSIVE OR
6F87:0A                    ASL  A           ; MOVE TO CARRY
6F88:0A                    ASL  A           ;
6F89:2E AB 6F              ROL  PSR1        ; SHIFT LOW PSR
6F8C:2E AC 6F              ROL  PSR2        ; SHIFT NEXT PSR
6F8F:2E AD 6F              ROL  PSR3        ; AND ONCE MORE
6F92:2E AE 6F              ROL  PSR4        ; FINALLY THE HIGH BYTE
6F95:88                    DEY              ; REPEAT FOR EVERY BIT IN BSIZE
6F96:D0 E6                 BNE  ADVANCE     ;

6F98:AD AB 6F      RANGE   LDA  PSR1        ; GET VALUE
6F9B:2D AF 6F              AND  BSIZE       ; MASK NEXT BINARY VALUE
6F9E:CD B1 6F              CMP  MODULO      ; IS VALUE TOO BIG?
6FA1:B0 D8                 BCS  REUSEN      ; YES, TRY AGAIN
6FA3:8D B2 6F              STA  HOLD        ; SAVE VALID PSR
6FA6:60                    RTS              ; AND EXIT
```

PROGRAM RM-7, CONT'D . . .

```
6FA7:          ;              *** PSR REGISTERS ***

6FA7:          ;   SEEDL AND SEEDH HOLD THE STARTING SEED SHOULD YOU
6FA7:          ;   WANT TO RERUN THE SERIES.  PSR1, PSR2, PSR3, AND
6FA7:          ;   PSR4 FORM THE 23 BIT PSEUDORANDOM SEQUENCER.
6FA7:          ;
6FA7:          ;   BSIZE IS A SIZING MASK.
6FA7:          ;
6FA7:          ;   MODULO HOLDS THE VALUE N, WHILE HOLD KEEPS THE RANDOM (N)
6FA7:          ;

6FA7:AA          SEED1    DFB   $AA       ; SEED LOW VALUE
6FA8:AA          SEED2    DFB   $AA       ; SEED SECOND LOWEST
6FA9:AA          SEED3    DFB   $AA       ; SEED THIRD LOWEST
6FAA:AA          SEED4    DFB   $AA       ; HIGH SEED
6FAB:AA          PSR1     DFB   $AA       ; PSR LOW BYTE
6FAC:AA          PSR2     DFB   $AA       ; PSR SECOND LOWEST
6FAD:3B          PSR3     DFB   $3B       ; PSR THIRD LOWEST
6FAE:AA          PSR4     DFB   $AA       ; PSR HIGHEST
6FAF:FF          BSIZE    DFB   $FF       ; SAVE OF BINARY SIZE
6FB0:04          RSIZE    DFB   $04       ; YSAVE FOR RETRY

6FB1:07          MODULO   DFB   $07       ; MAXIMUM SIZE OF N
6FB2:00          HOLD     DFB   $00       ; SAVE OF PSR VALUE
```

8

SHUFFLE

a fast ''random exchange'' method of rearranging cards or number arrays

There are lots of computer situations where you might like to take a pile of objects and rearrange them into some different order.

Shuffling a deck of cards is the most obvious example of this sort of thing. You might use playing cards for poker or blackjack simulations. Other times, the cards may have different symbols or messages on them. Tarot cards are an example, as are the *Chance* and *Community Chest* decks in a Monopoly simulation.

The things being shuffled need not be paper cards, of course. They could be tiles in a magic number game, letters in a word, the sequence in which new things appear, a maze in an adventure, or a journey into cryptography.

The fancy name for shuffling is *randomizing without replacement.* In randomizing without replacement, you simply rearrange a fixed array of values that you already have on hand. Once drawn from the deck, the four of clubs will not reappear.

The random number generator of the last ripoff module kept all the marbles in the pipe. You just cloned off the marbles you wanted. This was *randomizing with replacement.* In randomizing with replacement, the same value can come up over and over again.

Hence . . .

Randomizing WITH Replacement—

Grabbing a random number without removing that number from being available for future grabs.

Rolling a die is an example.

Randomizing WITHOUT Replacement—

Grabbing a random number while also eliminating the availability of that number for future grabs.

Shuffling cards is typical.

Note that these are totally different things. You'll get absurd results if you try to use the wrong one. Like only six different throws of a die before the die is "empty." Or the nine of spades dealt to you three cards in a row.

To throw some other terms at you, grabbing without replacement involves an *infinite pool* of numbers. Or at least an irrigation ditch full.

Grabbing with replacement involves a *finite pool* of numbers. These numbers are usually arranged into a fixed and rather small *array*. The array size on a playing card deck is usually 52.

The typical way that beginners try to shuffle things on their Apple has two very serious flaws. First, of course, they will be trying to use the Applesoft RND subroutine, which, as we have seen, is not.

Besides being rather slow.

We can easily fix this particular hassle by switching to the random number generator of the last ripoff module.

The second problem is more subtle. If you grab 52 random numbers in a row, you have to check each *new* number to make sure it was not duplicated before. This is no problem on the first card, and is trivial on the first few cards. But on, say card 50, the odds are 50/52 that you already have this card and have to go back again and again.

In fact, for your last card, you might need 52 *additional* tries to pick up only a 0.63 odds of finding the remaining card.

1/e and all that statistical stuff.

You, in fact, have to deal hundreds or even thousands of cards to be reasonably sure of getting 52 different ones. So, testing for duplicates is a bad scene because it takes ridiculously long and involves many wasted trips to the random number generator.

Let's work smarter and not harder. Do not try to take your numbers out of an infinite pool. Instead, take them out of a small and fixed array. Center your activities on rearranging the array.

Here is a good and fast way to shuffle a pile of something . . .

TO REARRANGE N OBJECTS—

Take the object in the first location and interchange it with an object in another location in the array, chosen at random.

Then take the object in the second location and do the same thing.

Repeat this for all the locations.

In other words, lay your 52 cards on the table. Grab the first card and interchange it with any card, picked at random. Next, grab the second card and interchange it with any card, again picked at random. Continue the process till you run out of cards.

Note two things. First, there are only 52 random numbers needed this way, since each random number gets used only once. Secondly, a card in some position will sometimes replace itself. This happens if the card in location number seven is interchanged with the random location number seven that just came up.

The odds on a card replacing itself are exactly the same as shuffling a real card deck and having the same card end up in the same position.

Which is rare but it certainly can happen. You can even get the deck back exactly the way you started. Odds on this are a tad low, though. The key point is that this random interchange method exactly duplicates a fair and thorough shuffle of real cards.

The same thing works for other shuffles. For a 15-tile magic square, you only interchange 15 values. You only swap six letters to jumble a six letter word, and so on.

Let's try it.

A Shuffler

The subroutine called SHUFFLR will take an array named CARDECK of length ARNUM and reorder everything.

SHUFFLR does this by using the random number generator of the previous ripoff module. CARDECK is presently set up to hold 52 cards, and ARNUM equals decimal 52 or hex $34.

The shuffling process is done by taking the first array value and interchanging it with an array value in a slot chosen at random. To find the exchange slot, you get a random number from 0 to 51 and do the exchange. The process gets repeated 52 times, thus swapping each card with some other card or itself at least once.

One of the array values being swapped is temporarily stashed on the stack. This handles the juggling process of moving two things between two locations without dropping either one of them.

To use SHUFFLR for other tasks, you just change the array values and the size of your array.

Note that SHUFFLR does not care what is inside each array slot. This lets you use meaningful codes for each array value. These codes are totally independent of the shuffling process.

How do you code a deck of cards?

One way to code the cards is to use one hex digit for the values from ace through king. An obvious choice is to use $X1 for an ace, $X2 for a two, $X9 for a nine, $XA for a ten, $XB for a jack, and so on. Let's use the least significant hex digit for this.

We will use the other hex digit to pick a suit. Say $0X for hearts, $1X for diamonds, $2X for clubs, and $3X for spades. Thus, the ace of spades will be coded $31, while the king of diamonds will be a $1D.

Clear?

A companion demo called DEALER will exercise your shuffler. The "S" key will shuffle the deck. The "R" key will repeat the previous shuffle for a replay. The "D" key, or optionally, the spacebar deals a card. The "Q" key quits the program for you.

We have used the short file printing method to handle text. It is the better choice here because we have lots of short and ordered words that we need in a more or less random access way. We are also under the nasty 256 character limit here. Use of a short file also saves dragging IMPRINT into the demo. It also gives you a chance to play with absolute indexed addressing.

The four options needed are simple enough that we will handle them by brute force, rather than using fancier option picker code.

We have also included a card counter. This one can be used for a position or a score, and does not need any hex-to-decimal or decimal-to-hex conversion. I'll leave it to you to puzzle out how this one works.

Naturally, a machine language randomizing-without-replacement module is far too fast to use as a real time playing card shuffle. So, we'll have to slow it down bunches. To do this, we will build the shuffler into a sound effect that mimics a deck being shuffled, and adjust the timing to "real" time.

Should you need something rearranged very quickly, be sure to defeat the sound effects. Or else adjust the effects to mimic what you are emulating.

Once again, the pseudo-random generator of the previous ripoff module is needed to get this module to work. So, be sure to have either RANDOM or THE WHOLE BALL OF WAX in your machine when using either the shuffler or the card demo.

MIND BENDERS

—What is the total time needed to shuffle a deck of 52 cards, with and without the sound effects?

—Add a HIRES or LORES graphics display of the playing cards.

—Show how to do an "instant solver" for those word jumble puzzles on a newspaper's comics page.

—In a word guessing game that has a file of hundreds of words, show how to get each word just once yet in a different order each session.

—Why does the card number display work in decimal without needing any hex conversion?

—What changes are needed to make the demo professionally useful?

PROGRAM RM-8
SHUFFLE

```
----- NEXT OBJECT FILE NAME IS SHUFFLER
7000:            ORG  $7000      ; PUT MODULE #8 AT $7000


7000:         ;  *****************************************
7000:         ;  *                                       *
7000:         ;  *           -< SHUFFLER >-              *
7000:         ;  *                                       *
7000:         ;  *  ( RANDOMIZING WITHOUT REPLACEMENT )  *
7000:         ;  *                                       *
7000:         ;  *     VERSION 1.0  ($7000-$7246)        *
7000:         ;  *                                       *
7000:         ;  *              5-24-83                  *
7000:         ;  *.......................................*
7000:         ;  *                                       *
7000:         ;  *         COPYRIGHT C 1983 BY           *
7000:         ;  *                                       *
7000:         ;  *     DON LANCASTER AND SYNERGETICS     *
7000:         ;  *     BOX 1300, THATCHER AZ., 85552     *
7000:         ;  *                                       *
7000:         ;  *     ALL COMMERCIAL RIGHTS RESERVED    *
7000:         ;  *                                       *
7000:         ;  *****************************************


7000:         ;           *** WHAT IT DOES ***

7000:         ;  THIS MODULE SHOWS YOU HOW TO SHUFFLE OR REARRANGE
7000:         ;  AN ARRAY OF CARDS, NUMBERS, LETTERS, OR OBJECTS.
7000:         ;


7000:         ;           *** HOW TO USE IT ***

7000:         ;  TO USE THE SHUFFLER:
7000:         ;
7000:         ;     START YOUR ARRAY FILE WITH CARDECK AT $7213.
7000:         ;     PUT THE NUMBER OF ARRAY ELEMENTS IN ARNUM AT $710E.
7000:         ;     THEN JSR SHUFFLR AT $70F1.  EQUIVALENT APPLESLOTH
7000:         ;     LOCATIONS ARE 29203, 28942, AND 28913.

7000:         ;  TO RUN THE CARD DEALER DEMO:
7000:         ;
7000:         ;     JSR DEALER AT $7000 OR CALL 28672.
```

PROGRAM RM-8, CONT'D . . .

```
7000:          ;           *** GOTCHAS ***

7000:          ;   THE RANDOM SUBROUTINE MUST BE PRESENT IN THE
7000:          ;   MACHINE.  PRELOAD "RANDOM" OR "THE WHOLE BALL
7000:          ;   OF WAX" TO DO THIS.
7000:          ;
7000:          ;   YOUR ARRAY FILE MUST BE PRELOADED WITH THE
7000:          ;   PROPER VALUES.


7000:          ;          *** ENHANCEMENTS ***

7000:          ;   WORDS, OBJECTS, OR OTHER TYPES OF CARDS ARE DONE
7000:          ;   BY CHANGING THE MEANING AND SIZE OF YOUR ARRAY.
7000:          ;


7000:          ;          *** RANDOM COMMENTS ***

7000:          ;   THIS SHUFFLE DEMO IS INTENDED TO SHOW THE PROCESS
7000:          ;   INVOLVED.  AN ACTUAL CARD PROGRAM HAS TO BE FAR
7000:          ;   FRIENDLIER THAN THIS, AND SHOULD DISPLAY REAL CARDS.
7000:          ;
7000:          ;   THE DEMO ALSO SHOWS HOW TO HANDLE SIMPLE SCORING
7000:          ;   WITHOUT NEEDING HEX TO DECIMAL CONVERSION.
```

PROGRAM RM-8, CONT'D . . .

```
7000:          ;                *** HOOKS ***

FDF0:          COUT1    EQU  $FDF0      ; OUTPUT TEXT TO SCREEN
FC58:          HOME     EQU  $FC58      ; CLEAR TEXT SCREEN AND HOME CURSOR
FB2F:          INIT     EQU  $FB2F      ; INITIALIZE TEXT SCREEN
C000:          IOADR    EQU  $C000      ; KEYBOARD INPUT LOCATION
C010:          KBDSTRB  EQU  $C010      ; KEYBOARD STROBE RESET
FD1B:          KEYIN    EQU  $FD1B      ; MONITOR READKEY SUBROUTINE
FE80:          SETINV   EQU  $FE80      ; SET INVERSE SCREEN
FE84:          SETNORM  EQU  $FE84      ; SET NORMAL SCREEN
C030:          SPKR     EQU  $C030      ; SPEAKER CLICK OUTPUT
FCA8:          WAIT     EQU  $FCA8      ; MONITOR TIME DELAY

6F5B:          RANDOM   EQU  $6F5B      ; RANDOM NUMBER INITIALIZER
6F2E:          RESEED   EQU  $6F2E      ; RANDOM NUMBER SEEDER
6F7B:          REUSEN   EQU  $6F7B      ; RANDOM NUMBER GENERATOR

0020:          WNDLFT   EQU  $20        ; LEFT SIDE OF SCROLL WINDOW
0021:          WNDWTH   EQU  $21        ; WIDTH OF SCROLL WINDOW
0022:          WNDTOP   EQU  $22        ; TOP OF SCROLL WINDOW
0023:          WNDBTM   EQU  $23        ; BOTTOM OF SCROLL WINDOW
0024:          CH       EQU  $24        ; CURSOR HORIZONTAL POSITION
0033:          PROMPT   EQU  $33        ; PROMPT SYMBOL


7000:          ;        *** TEXTFILE COMMANDS ***

0088:          B        EQU  $88        ; BACKSPACE
008D:          C        EQU  $8D        ; CARRIAGE RETURN
0084:          D        EQU  $84        ; DOS ATTENTION
009B:          E        EQU  $9B        ; ESCAPE
008A:          L        EQU  $8A        ; LINEFEED
0060:          P        EQU  $60        ; FLASHING PROMPT
0000:          X        EQU  $00        ; END OF MESSAGE
```

PROGRAM RM-8, CONT'D . . .

```
7000:              ;       *** DEALIN DEMO ***

7000:              ;    THIS DEMO EXCERCISES THE SHUFFLER
7000:              ;    ON A STANDARD DECK OF 52 CARDS.


7000:20 2F FB    DEALER  JSR   INIT        ; SET UP TEXT SCREEN
7003:20 58 FC            JSR   HOME        ; AND CLEAR IT
7006:20 2E 6F            JSR   RESEED      ; RESEED RANDOM
7009:AD 0E 71            LDA   ARNUM       ; GET ARRAY NUMBER
700C:20 5B 6F            JSR   RANDOM      ; INIT RANDOM

700F:A9 07               LDA   #$07        ; TAB 7 TO RIGHT
7011:85 24               STA   CH          ;
7013:20 80 FE            JSR   SETINV      ; INVERSE TITLE
7016:A0 5E               LDY   #>MS0-CV1   ; GET HEADER
7018:20 E3 70            JSR   TEXT8       ; AND DISPLAY
701B:20 84 FE            JSR   SETNORM     ; NORMAL TEXT
701E:A0 74               LDY   #>MS1-CV1   ; GET SCREEN PROMPTS
7020:20 E3 70            JSR   TEXT8       ; AND DISPLAY

7023:A9 07               LDA   #$07        ; SET LOWSCREEN WINDOW
7025:85 22               STA   WNDTOP      ;
7027:A9 05               LDA   #$05        ;
7029:85 20               STA   WNDLFT      ; TAB OVER TO CENTER
702B:A9 22               LDA   #$22        ;
702D:85 21               STA   WNDWTH      ;
702F:20 58 FC            JSR   HOME        ; GET INSIDE WINDOW

7032:2C 10 C0    CMND8   BIT   KBDSTRB     ; RESET KEYBOARD
7035:AD 00 C0    LOOK8   LDA   IOADR       ; READ KEYBOARD
7038:10 FB               BPL   LOOK8       ;
703A:2C 10 C0            BIT   KBDSTRB     ;
703D:C9 E1               CMP   #$E1        ; FORCE CASE
703F:90 02               BCC   CSORT       ;
7041:E9 20               SBC   #$20        ; SUBTRACT TO CHANGE CASE

7043:C9 D3       CSORT   CMP   #$D3        ; S FOR SHUFFLE?
7045:F0 1D               BEQ   SHUFF       ; YES
7047:C9 C4               CMP   #$C4        ; D FOR DEAL ?
7049:F0 2E               BEQ   DEAL        ;
704B:C9 A0               CMP   #$A0        ; ALSO SPACE FOR DEAL
704D:F0 2A               BEQ   DEAL        ;
704F:C9 D2               CMP   #$D2        ; R FOR REPLAY?
7051:F0 17               BEQ   REPLAY      ;
7053:C9 D1               CMP   #$D1        ; Q FOR QUIT?
7055:F0 06               BEQ   QUIT8       ;
7057:20 18 71            JSR   EFFECT2     ; BLORK
705A:4C 32 70            JMP   CMND8       ; TRY AGAIN FOR LEGAL KEY
```

PROGRAM RM-8, CONT'D . . .

```
705D:           ;       *** QUIT EXIT ***

705D:20 2F FB   QUIT8   JSR  INIT       ; OPEN WINDOW
7060:20 58 FC           JSR  HOME       ; CLEAR SCREEN
7063:60                 RTS             ; AND EXIT ON "Q"


7064:           ;       *** SHUFF PROCESSING ***


7064:           ;      THIS CODE SHUFFLES THE DECK AND
7064:           ;      RESETS THE CARD COUNTERS TO ONE.


7064:20 F1 70   SHUFF   JSR  SHUFFLR    ; SHUFFLE THE DECK
7067:4C 6A 70           JMP  REPLAY     ; RESET COUNTERS



706A:           ;      *** REPLAY MODULE ***

706A:           ;     RESETS THE CARD COUNTER TO ZERO.


706A:A0 00      REPLAY  LDY  #$00       ; RESET COUNTERS
706C:8C EF 70           STY  HEXCNT     ;
706F:C8                 INY             ; ONE MORE FOR PEOPLE
7070:8C F0 70           STY  DECCNT     ;
7073:20 58 FC           JSR  HOME       ; CLEAR OLD CARDS
7076:4C 32 70           JMP  CMND8      ; GO GET NEXT COMMAND
```

PROGRAM RM-8, CONT'D . . .

```
7079:               ;       *** DEAL PROCESSING ***

7079:               ;    THIS CODE TRYS TO DEAL A CARD IF
7079:               ;    THERE ARE ANY LEFT IN THE DECK.

7079:AD EF 70       DEAL     LDA  HEXCNT     ; GET NUMBER IN DECK (52)
707C:CD 0E 71                CMP  ARNUM      ; ANY CARDS LEFT?
707F:B0 5A                   BCS  EMPTY8     ; NO, SAY SO

7081:A0 A5                   LDY  #>MS2-CV1  ; SAY "CARD"
7083:20 E3 70                JSR  TEXT8      ;
7086:AD F0 70                LDA  DECCNT     ; GET TENS FOR CARD NUMBER
7089:4A                      LSR  A          ;  AND SHIFT FOUR TO RIGHT
708A:4A                      LSR  A          ;
708B:4A                      LSR  A          ;
708C:4A                      LSR  A          ;
708D:F0 05                   BEQ  LOWDEC     ; IS IT NONZERO?
708F:09 B0                   ORA  #$B0       ; CHANGE TO ASCII
7091:20 F0 FD                JSR  COUT1      ; AND PRINT IT
7094:AD F0 70       LOWDEC   LDA  DECCNT     ; GET UNITS FOR CARD NUMBER
7097:29 0F                   AND  #$0F       ; MASK TENS
7099:09 B0                   ORA  #$B0       ; CHANGE TO ASCII
709B:20 F0 FD                JSR  COUT1      ; AND PRINT IT

709E:A0 AB                   LDY  #>MS3-CV1  ; SAY "IS THE"
70A0:20 E3 70                JSR  TEXT8      ;  TO SCREEN

70A3:AE EF 70                LDX  HEXCNT     ; GET CARD
70A6:BD 13 72                LDA  CARDECK,X  ; FROM DECK
70A9:48                      PHA             ; AND SAVE FOR SUIT
70AA:29 0F                   AND  #$0F       ; MASK SUIT
70AC:AA                      TAX             ; USE AS INDEX
70AD:CA                      DEX             ; MAKE ACE=1, NOT ZERO!
70AE:BC 2A 71                LDY  CARVAL,X   ; GET SUIT NAME
70B1:20 E3 70                JSR  TEXT8      ;  AND PRINT TO SCREEN

70B4:A0 B4                   LDY  #>MS4-CV1  ; SAY "OF"
70B6:20 E3 70                JSR  TEXT8      ;   AND PRINT IT

70B9:68                      PLA             ; GET CARD BACK
70BA:4A                      LSR  A          ; AND SHIFT TO RIGHT
70BB:4A                      LSR  A          ;
70BC:4A                      LSR  A          ;
70BD:4A                      LSR  A          ;
70BE:AA                      TAX             ; USE AS INDEX
70BF:BC 37 71                LDY  CARSUIT,X  ; GET SUIT NAME
70C2:20 E3 70                JSR  TEXT8      ; AND PRINT IT
```

PROGRAM RM-8, CONT'D . . .

```
70C5:A0 B9                LDY  #>MS5-CV1 ; GET PERIOD AND CR
70C7:20 E3 70             JSR  TEXT8     ; AND PRINT IT
70CA:EE EF 70             INC  HEXCNT    ; GO TO NEXT CARD
70CD:F8                   SED            ; GO TO DECIMAL FOR SCORE
70CE:18                   CLC            ;
70CF:AD F0 70             LDA  DECCNT    ; INCREMENT DECIMAL
70D2:69 01                ADC  #$01      ;
70D4:8D F0 70             STA  DECCNT    ;
70D7:D8                   CLD            ; GET OUT OF DECIMAL!
70D8:4C 32 70             JMP  CMND8     ; GO GET NEXT COMMAND

70DB:A0 BD         EMPTY8 LDY  #>MS6-CV1 ; GET EMPTY MESSAGE
70DD:20 E3 70             JSR  TEXT8     ; PUT ON SCREEN
70E0:4C 32 70             JMP  CMND8     ; GO GET NEXT COMMAND


70E3:              ;       *** TEXT GENERATOR ***


70E3:              ;    THIS USES THE SHORT FILE METHOD TO PUT
70E3:              ;    MESSAGES ONLY ON THE SCREEN.


70E3:B9 3B 71      TEXT8  LDA  CV1,Y     ; GET NEXT CHARACTER
70E6:F0 06                BEQ  DONE8     ; TEST FOR $00
70E8:20 F0 FD             JSR  COUT1     ; OUTPUT TO SCREEN
70EB:C8                   INY            ; GO TO NEXT CHARACTER
70EC:D0 F5                BNE  TEXT8     ; AND REPEAT

70EE:60            DONE8  RTS            ; RETURN WHEN FINISHED



70EF:              ;        *** CARD COUNTER STASH ***


70EF:00            HEXCNT DFB  $00       ; HEX COUNT FOR MACHINE
70F0:01            DECCNT DFB  $01       ; DECIMAL COUNT FOR PEOPLE
```

PROGRAM RM-8, CONT'D . . .

```
70F1:              ;        *** SHUFFLER SUBROUTINE ***


70F1:              ;    THIS MODULE REARRANGES THE ARRAY CALLED CARDECK
70F1:              ;    AND WHOSE LENGTH IS STORED IN ARNUM.
70F1:              ;
70F1:              ;    THE RANDOM SUBROUTINE MUST BE PRESENT IN THE
70F1:              ;    MACHINE AND MUST BE PREVIOUSLY SEEDED AND
70F1:              ;    INITIALIZED.


70F1:AE 0E 71      SHUFFLR LDX  ARNUM      ; GET NUMBER OF SWAPS
70F4:CA                    DEX             ; FOR ARRAY 0-51, NOT 1-52
70F5:20 7B 6F      NEXT8   JSR  REUSEN     ; GET RANDOM POSITION
70F8:A8                    TAY             ; AND HOLD IN Y REGISTER
70F9:BD 13 72              LDA  CARDECK,X  ; GET FIRST FIXED VALUE
70FC:48                    PHA             ; STASH TO JUGGLE
70FD:B9 13 72              LDA  CARDECK,Y  ; GET RANDOM NEXT VALUE
7100:9D 13 72              STA  CARDECK,X  ; REPLACE NEXT WITH FIRST
7103:68                    PLA             ; JUGGLE BACK
7104:99 13 72              STA  CARDECK,Y  ; REPLACE FIRST WITH NEXT
7107:20 0F 71              JSR  EFFECT1    ; MAKE NOISE (OPTIONAL)
710A:CA                    DEX             ; ONE LESS POSITION
710B:10 E8                 BPL  NEXT8      ; REPEAT FOR EACH POSITION
710D:60                    RTS             ; QUIT WHEN FINISHED



710E:              ;        *** SHUFFLER STASH ***

710E:34            ARNUM   DFB  52         ; NUMBER OF ELEMENTS IN ARAY



710F:              ;        *** SHUFFLER SOUND EFFECTS ***

710F:8A            EFFECT1 TXA             ; DECK SHUFFLING SOUND
7110:D0 02                 BNE  NOZERO8    ; DISALLOW ZERO VALUE
7112:A9 01                 LDA  #$01       ;
7114:0A            NOZERO8 ASL  A          ; SLOW IT DOWN!
7115:20 A8 FC              JSR  WAIT       ; DELAY, THEN FALL THROUGH
7118:A9 05         EFFECT2 LDA  #$05       ; NUMBER OF CLICKS PER WHAP
711A:48            NEXTWP  PHA             ; SAVE ON STACK
711B:2C 30 C0              BIT  SPKR       ; MOVE SPEAKER CONE
711E:A9 07                 LDA  #$07       ; SET PITCH OF WHAP
7120:20 A8 FC              JSR  WAIT       ;
7123:68                    PLA             ; GET CLICK COUNTER
7124:38                    SEC             ;
7125:E9 01                 SBC  #$01       ; AND COUNT DOWN
7127:D0 F1                 BNE  NEXTWP     ;
7129:60                    RTS             ; AND RETURN
```

PROGRAM RM-8, CONT'D . . .

```
712A:          ;       *** MESSAGE FILE ***


712A:          ;  WE'LL USE THE SHORT FILE METHOD HERE SINCE
712A:          ;  RANDOM ACCESS OF A FEW SHORT AND FIXED
712A:          ;  MESSAGES ARE NEEDED.
712A:          ;


712A:          ;      *** MESSAGE POINTERS ***


712A:00        CARVAL  DFB  >CV1-CV1   ; THESE POINT TO CARD VALUES
712B:04                DFB  >CV2-CV1   ;
712C:08                DFB  >CV3-CV1   ;
712D:0E                DFB  >CV4-CV1   ;
712E:13                DFB  >CV5-CV1   ;
712F:18                DFB  >CV6-CV1   ;
7130:1C                DFB  >CV7-CV1   ;
7131:22                DFB  >CV8-CV1   ;
7132:28                DFB  >CV9-CV1   ;
7133:2D                DFB  >CV10-CV1 ;
7134:31                DFB  >CV11-CV1 ;
7135:36                DFB  >CV12-CV1 ;
7136:3C                DFB  >CV13-CV1 ;

7137:41        CARSUIT DFB  >CS0-CV1   ; THESE POINT TO THE CARD SUITS
7138:48                DFB  >CS1-CV1   ;
7139:51                DFB  >CS2-CV1   ;
713A:57                DFB  >CS3-CV1   ;
```

PROGRAM RM-8, CONT'D . . .

```
713B:           340 ;     *** THE CARD VALUES ***

713B:C1 C3 C5   342 CV1      ASC              "ACE"
713E:00         343          DFB              X

713F:D4 D7 CF   345 CV2      ASC              "TWO"
7142:00         346          DFB              X

7143:D4 C8 D2   348 CV3      ASC              "THREE"
7146:C5 C5
7148:00         349          DFB              X

7149:C6 CF D5   351 CV4      ASC              "FOUR"
714C:D2
714D:00         352          DFB              X

714E:C6 C9 D6   354 CV5      ASC              "FIVE"
7151:C5
7152:00         355          DFB              X

7153:D3 C9 D8   357 CV6      ASC              "SIX"
7156:00         358          DFB              X

7157:D3 C5 D6   360 CV7      ASC              "SEVEN"
715A:C5 CE
715C:00         361          DFB              X

715D:C5 C9 C7   363 CV8      ASC              "EIGHT"
7160:C8 D4
7162:00         364          DFB              X

7163:CE C9 CE   366 CV9      ASC              "NINE"
7166:C5
7167:00         367          DFB              X

7168:D4 C5 CE   369 CV10     ASC              "TEN"
716B:00         370          DFB              X

716C:CA C1 C3   372 CV11     ASC              "JACK"
716F:CB
7170:00         373          DFB              X

7171:D1 D5 C5   375 CV12     ASC              "QUEEN"
7174:C5 CE
7176:00         376          DFB              X

7177:CB C9 CE   378 CV13     ASC              "KING"
717A:C7
717B:00         379          DFB              X
```

PROGRAM RM-8, CONT'D . . .

```
717C:          382 ;     *** THE CARD SUITS ***


717C:C8 C5 C1  384 CS0     ASC               "HEARTS"
717F:D2 D4 D3
7182:00        385         DFB               X

7183:C4 C9 C1  387 CS1     ASC               "DIAMONDS"
7186:CD CF CE
7189:C4 D3
718B:00        388         DFB               X

718C:C3 CC D5  390 CS2     ASC               "CLUBS"
718F:C2 D3
7191:00        391         DFB               X

7192:D3 D0 C1  393 CS3     ASC               "SPADES"
7195:C4 C5 D3
7198:00        394         DFB               X




7199:          396 ;     *** TEXT SCREEN MESSAGES ***


7199:C3 C1 D2  398 MS0     ASC               "CARD SHUFFLING DEMO"
719C:C4 A0 D3
719F:C8 D5 C6
71A2:C6 CC C9
71A5:CE C7 A0
71A8:C4 C5 CD
71AB:CF
71AC:8D 8D 00  399         DFB               C,C,X

71AF:A8 D3 A9  401 MS1     ASC               "(S)HUFFLE, (D)EAL, (R)EPLAY, (Q)UIT ?
71B2:C8 D5 C6
71B5:C6 CC C5
71B8:AC A0 A8
71BB:C4 A9 C5
71BE:C1 CC AC
71C1:A0 A8 D2
71C4:A9 C5 D0
71C7:CC C1 D9
71CA:AC A0 A8
71CD:D1 A9 D5
71D0:C9 D4 A0
71D3:BF
71D4:8D 8D     402         DFB               C,C
71D6:AD AD AD  403         ASC               "---> "
71D9:BE A0
71DB:60 88 8D  404         DFB               P,B,C,C,X
71DE:8D 00
```

PROGRAM RM-8, CONT'D . . .

```
71E0:C3 C1 D2  407 MS2     ASC             "CARD "
71E3:C4 A0
71E5:00        408         DFB             X

71E6:A0 C9 D3  410 MS3     ASC             " IS THE "
71E9:A0 D4 C8
71EC:C5 A0
71EE:00        411         DFB             X

71EF:A0 CF C6  413 MS4     ASC             " OF "
71F2:A0
71F3:00        414         DFB             X

71F4:AE        416 MS5     ASC             "."
71F5:8D 8D 00  417         DFB             C,C,X

71F8:A0 A0 A0  419 MS6     ASC             "   SORRY, DECK IS EMPTY!"
71FB:D3 CF D2
71FE:D2 D9 AC
7201:A0 C4 C5
7204:C3 CB A0
7207:C9 D3 A0
720A:C5 CD D0
720D:D4 D9 A1
7210:8D 8D 00  420         DFB             C,C,X
```

PROGRAM RM-8, CONT'D . . .

```
7213:           423 ;       *** DECK OF CARDS ***

7213:           425 ;  THE LOW BYTE OF EACH ENTRY IS THE CARD
7213:           426 ;  VALUE WITH X1=ACE, X2=TWO, XA=TEN, ETC.
7213:           427 ;
7213:           428 ;  THE HIGH BYTE OF EACH ENTRY IS THE CARD
7213:           429 ;  SUIT WITH 0X=HEARTS, 1X=DIAMONDS, 2X=
7213:           430 ;  CLUBS, AND 3X=SPADES.


7213:01 02 03  432 CARDECK DFB            $01,$02,$03,$04,$05,$06,$07
7216:04 05 06
7219:07
721A:08 09 0A  433         DFB            $08,$09,$0A,$0B,$0C,$0D
721D:0B 0C 0D

7220:11 12 13  435         DFB            $11,$12,$13,$14,$15,$16,$17
7223:14 15 16
7226:17
7227:18 19 1A  436         DFB            $18,$19,$1A,$1B,$1C,$1D
722A:1B 1C 1D

722D:21 22 23  438         DFB            $21,$22,$23,$24,$25,$26,$27
7230:24 25 26
7233:27
7234:28 29 2A  439         DFB            $28,$29,$2A,$2B,$2C,$2D
7237:2B 2C 2D

723A:31 32 33  441         DFB            $31,$32,$33,$34,$35,$36,$37
723D:34 35 36
7240:37
7241:38 39 3A  442         DFB            $38,$39,$3A,$3B,$3C,$3D
7244:3B 3C 3D



*** SUCCESSFUL ASSEMBLY: NO ERRORS
```

APPENDIX A

DIFFERENCES BETWEEN "OLD" AND "NEW" EDASM

Apple Computer's EDASM editor/assembler has recently been overhauled and upgraded. There are now *two* new versions, one for DOS 3.3e, and one for ProDOS. Both are available in their respective toolkits in Apple's *Workbench* series.

The bottom line is that EDASM is now a first class, first rate macroassembler with just about all the bells and whistles anyone could ask for, including dual file editing, "library" module insertion, in-place assembly, and co-resident assembly. While the editor portion of EDASM remains as putrid as ever, you can simply use Applewriter IIe and WPL instead, doing "new way" editing as in chapter five.

Included with either EDASM is a new debugging tool called the BUGBYTER. The BUGBYTER includes a fancy upgrade of the old miniassembler, along with greatly improved single step, trace, and debug routines in a package that lets you do much more and do it much more quickly. For instance, there are single keystrokes to pick any screen mode, and you can use the game paddle to control debugging speed. You can even debug parts of your code at full speed and parts at slow speed. This is most handy for time critical routines. BUGBYTER is runnable anywhere in memory.

Very little was lost in going from "old" EDASM to "new" EDASM. With "new" EDASM, reserved labels normally include "A," "a," "X," "x," "Y," and "y," instead of just "A." You also *must* separate the op code and operand of SKP, ROL, ROR, ASL, LRS, and LST. Thus, a "SKP5" or a "ROLA" command will generate error messages under "new" EDASM. To use "old" EDASM source code with "new"

EDASM, spaces must be added between op code and operand of all these commands. Tab settings are also different in the "new" version.

Here's a list of the important changes and improvements to the *editor* portion of DOS 3.3e version of "new" EDASM. Note that anything you don't like about these features is easily gotten around by doing "new way" editing under Applewriter IIe instead.

Here goes:

1. The author of "old" EDASM was Randy Wigginton; the new author is John Arkley, who upgraded and improved Randy's original work.
2. The BUGBYTER is included, a tremendous improvement over the old miniassembler, single step, and trace routines.
3. System ID routines are now supplied and standardized, letting you configure your code for a II, II+, or IIe.
4. The work buffer is now 26,000 characters long, which is somewhat shorter than "old" EDASM. However, with "new way" editing under Applewriter IIe, your edit file can be 48,000 characters long.
5. The ASMIDSTAMP is restricted in its form so that real time clocks can be supported.
6. The FILE command now displays the slot and drive.
7. The manuals are greatly improved and now include tutorials.
8. The command level now automatically accepts either upper or lower case.
9. Combined upper and lower case is now standard on the Apple IIe. On older Apples, new commands of SETL and SETU are available for those Apples with a shift key mod and a lower case display. Commands of [E] (shift to lower case) and [W] (shift to upper case) are available for very old Apples without lower case. The screen will not be legible in lower case on these older machines.
10. Direct DOS commands using the "." prefix are not filtered for possible damage. In particular, ".SAVE" will plow the works.
11. You still cannot insert into the middle of your source code using "old way" editing. You have to use APPEND and then COPY. With "new way" editing, you can, of course, insert anything you want any place you want any time you want.
12. There is a new VOL command that goes along with SLOT and DRIVE that will return the current disk volume in use.
13. There is a new ADD command that lets you add text beyond a certain line number. Thus ADD 16 will add new lines *beyond* old line 16, compared to INS 16 which would insert new lines *before* old line 16.
14. The INSert or ADD modes can now be stopped with either a [D] or [Q].
15. A new REPLACE mode erases and then overwrites in one step. Before you had to DELete and then INSert.
16. There now is a recovery procedure to undo the NEW command. It is hairy to use, but it does exist.
17. A command of L43-6 lists six lines starting at line 43. Any time the second number is less than the first one, it is interpreted as "how many?".
18. The [R] command will relist whatever you last asked of [L].
19. A new command of SETD lets you change the delimiter from a ":". This lets you search and replace on a colon. Space or carriage returns are not allowed as delimiters.
20. You can now edit on both a range of numbers and a search string.
21. You can edit two files at once. The command of SWAP moves the two files between the "active" and "passive" editing buffers, sort of like a [Y] split screen in Applewriter IIe. The command of KILL2 deletes the "passive" buffer, similar to a "[Y]-N" in Applewriter IIe.
22. You can pick either 40 or 80 column operation with a "COL 40" or "COL 80" command.

23. There is now a simple way to undo the END command. Just set MAXFILES 5 and Call 3075.

Here are the major improvements in the *assembly* portion of "new" EDASM:

1. The DOS 3.3 version of "new" EDASM will not do an assembler listing to disk. You have to use the ProDOS version if you want to capture your normally printed assembler listing as a disk text file.
2. The trailer on an assembler listing now includes the date, line count, and remaining free space.
3. An "@" following an ASM command will suppress object code generation. This is handy for "quick looks" and finding potential errors.
4. The ASMIDSTAMP is no longer essential. On "old" EDASM, a FILE NOT FOUND error message was generated.
5. You can single step the assembly process by pressing the spacebar. Repeated spacebar hits do one line at a time. Pressing [ESC] on a 40-column screen lets you see the right half of the screen, or else switches back to the left half. Any other letter key resumes assembly at full speed.
6. Two *direct* keyboard commands override any imbedded LST ON or LST OFF commands. Use [N] to stop the listing, [O] to continue it.
7. The assembler will accept the tab key, [I], or the spacebar to enter a tab. This greatly eases the "tab problem" with "new way" editing.
8. The label in the label field is now called an IDENTIFIER.
9. The "a," "X," "x," "Y," and "y" labels are now reserved, in addition to "A." You can go to a lot of trouble to defeat this reservation if you have to. Good practice would also tell you to reserve "P," "p," "S," and "s" as well.
10. Macros are now available. These are disk based and are inserted when and as needed. Parameters can be passed back and forth between source code and macro.
11. A new operand of "*" is available that uses the present assembler program counter location. Intended use is to set aside specific positions in a page of memory. This can also be used to "pad" your way up to the next even page boundary.
12. You can now generate an absolute reference to a page zero location. To do this, put the EQU *after* the place in the source code where it first is needed. This is handy when you want to force an absolute long load, store, or whatever from an address on page zero, because of timing or code length considerations.
13. An upgraded OBJ command lets you assemble directly into the machine, without assembling to disk first. Tests are made to make sure there is no conflict with the assembly code itself. The combination of an "OBJ" command with an "ASM @" will directly assemble code into memory without generating any listing.
14. A new SW16 command will accept "Sweet 16" mnemonics. Three new commands have also been added to the original Sweet 16, which is a 16-bit pseudo interpreter. A compare, long branch, and subroutine long branch are now available. Use of Sweet 16 is usually shorter and simpler, but slower than doing your own custom 16-bit routines. One source of the new Sweet 16 code is EDASM itself. Just tear it apart using the "tearing method" of *Enhancing Your Apple II and IIe, Volume I,* (Sams 21822).
15. An undocumented X6502 command will apparently accept 65C02 mnemonics and, presumably, 65XC16 mnemonics as well. This command appeared in the preliminary documentation with a "we don't support this" disclaimer, but was dropped completely in the final manual.
16. New commands of ZDEF, ZREF, and ZXTRN are available that are

extensions of DEF. These forward-looking features require a linking loader that is not yet supported.

17. A new STR command works like ASC, only it includes a byte counter as its first character. Thus ASC gives you a text message, while STR gives you a text message preceded by the number of actual characters in the message.
18. A new DATE command reads the nine ASCII values stored at $03B8-$03C0 and enters them into the object code being generated. These locations usually hold the date portion of the ASMIDSTAMP.
19. A new IDNUM command reads the six ASCII values stored at $03C3 through $03C8 and enters them into the object code being generated. These locations usually hold the identity portion of the ASMIDSTAMP.
20. Conditional assembly has undergone a major overhaul. New commands of IFNE (not equal), IFEQ (equal), IFLT (less than), IFLE (less than or equal), IFGT (greater than), and IFGE (greater than or equal) are now available. A command of FAIL is also available for printing error messages.
21. A space must separate the op code and the operand on the SKP and LST commands.
22. Logical operators are now available, using the "↑" symbol for AND, "|" for OR, and "!" for EXOR. These operators work *only* on 16-bit arguments.
23. A new INCLUDE command stops the main assembly, assembles a source code module off disk, and then picks back up on the main assembly. This is most handy for inserting "mix and match" stock library routines.
24. Two commands of SBUFSIZ and IBUFSIZ let you adjust the size of your work areas for the original source code and the INCLUDE library module. See the manual for details. Changing buffer sizes is not normally needed.
25. A new MACLIB command tells the assembler that any "illegal" mnemonics are really the names of macro routines. Each macro routine is automatically done as if it was an INCLUDE command.
26. The "formfeed bug" has presumably been fixed, but it is still a good idea to force your own page breaks using the PAGE command.
27. A special column is available on the assembly listing to show branch destination addresses. Execution cycle times can also be optionally shown.
28. There are all sorts of new LST options. You can now separately turn off or on display of execution cycle times (C), generated object code (G), warnings (W), unassembled source code from bypassed conditional assembly (U), macro statements (E), alphabetic symbol tables (A), numeric symbol tables (V), or "six-across" symbol listings (S).
29. Standard tabbing values are different from "old" EDASM. Default tabs are now 16, 22, and 36, instead of 14, 19, 29. More than 80 columns may be needed for all the listing features and long comments. The simplest way to handle this is with 12 pitch on a daisywheel printer, or else use your own custom and "tighter" tab values. HINT: Keep your comments shorter than you did with "old" EDASM. This will help a lot.
30. New macro commands of "&0" and "&X" are available that control passing of parameters from the main source code to the macros. "&0" tells the number of parameters present in the operand field of the calling statement. "&X" keeps track of the number of times a macro is used. This allows the creation of local labels.
31. You can do co-resident assembly in a 64K Apple IIe, where the editor and assembler modules stay in the machine at the same time. An "*" following the ASM command will get the source file out of your machine, rather than off disk. This greatly speeds up the edit-assemble-

test round trip process. On short programs in certain areas of your machine, you can do both co-resident and in-place assembly at the same time. There are restrictions: You cannot use chaining, insertion, or macros when doing this, and your source code in the machine will get overwritten.

Finally, here are the differences between the ProDOS and DOS 3.3e versions of EDASM:

1. The ProDOS buffer is 37,000 characters long.
2. The ASMIDSTAMP is severely restrictive. It must be in DD-MM-YY format for clock compatibility.
3. A blank SBTL line still gets you the date.
4. The PFX command reads the current prefix. As is typical in ProDOS, a CAT command gets you a 40 column catalog, while the CATALOG command gives you all 80 columns. The CREATE command will generate a sub-directory.
5. The TYPE command lets you edit certain other file types, rather than just text files. You can also BLOAD, BSAVE, XLOAD, and XSAVE non-text files. The SYS command changes the type of source code file.
6. The EXIT command returns you to ProDOS BASIC. Commands of PTON and PTOFF turn the printer off and on, while EXEC will do a supervisory routine.
7. Time and date are automatically inset if a clock card is present. A TIME command is supported.
8. You can no longer do co-resident assembly. Preliminary ProDOS documentation did not support macros. Editing of two files at once also may not be supported.
9. You can route an assembler listing to diskette, instead of to printer, by using a "PR#6,ZORCHFILE" command.
10. There is a PAUSE command available to temporarily hold up assembly.
11. The error message on an aborted assembly is completely useless.

I personally despise ProDOS. Why? Because it is so unconscionably bloated, so user vicious, so buggy, and so incredibly poorly written. Nonetheless, if you must make an EDASM disk-based assembler listing (for "camera ready" print quality, typesetting, insertions, etc.), you will have to use ProDOS. The procedure is to take your DOS 3.3e text file, convert it with CONVERT, assemble to disk under ProDOS, and then CONVERT it back to the sane world.

Sigh.

Both ProDOS itself and the "new" versions of EDASM have numerous bugs in them. We will pass them on to you as we find out more about them.

Several specific bugs for now: The ProDOS routine of CONVERT can sometimes destroy a DOS 3.3e diskette. Seems a sector counter doesn't get incremented properly. Long filenames will often cause assembly problems. If it does not feel too much like assembling something, the ProDOS version of EDASM will simply kick sand in your face, instead of telling you what went wrong. That "ASSEMBLY ABORTED: LINE 0" message sure is friendly and helpful.

On either "new" version of EDASM, you will get error messages on a SKP5 or a LSTOFF, or an ASLA, and other places where "old" EDASM let you skip the space between op code and operand. Unfortunately, I did this just about everywhere in this book. Correcting the printed listings would most likely cause more grief than it would solve.

So, we have instead corrected all of the source code on the companion diskette.

Just remember to be sure and separate *all* op codes and operands with a space on "new" EDASM, and you should not have too much trouble.

Let us know about any other bugs as soon as you can.

APPENDIX B

SOME NAMES AND NUMBERS

ANTHRO DIGITAL SYSTEMS
Box 1385
Pittsfield, MA 01202
(413) 448-8278

APPLE ASSEMBLY LINE
Box 280300
Dallas, TX 75288
(214) 324-2050

APPLE AVOCATION ALLIANCE
721 Pike Street
Cheyenne, WY 82001
(307) 632-8581

A.P.P.L.E.
~~304 Main South~~ 21246 68th Ave S
~~Renton, WA 98055~~ Kent WA, 98032
~~(206) 271-4513~~ (206) 872-2245

APPLE COMPUTER
10260 Bandley Drive
Cupertino, CA 95014
(408) 996-1010

AVOCET SYSTEMS
804 South State Street
Dover, DE 19901
(302) 734-0151

BYTE
70 Main Street
Peterborough, NH 03458
(603) 924-9281

CENTRAL POINT SOFTWARE
Box 19730
Portland, OR 97219
(503) 244-5782

COMPUTER SHOPPER
Box F
Titusville, FL 32780
(305) 269-3211

CREATIVE COMPUTING
Box 789-M
Morristown, NJ 07960
(201) 540-0445

DENVER APPLE PI
Box 14767
Denver, CO 80217
(303) 429-4436

DECISION SYSTEMS
Box 13006
Denton, TX 76203
(817) 382-6353

DIABLO SYSTEMS
24500 Industrial Blvd.
Hayward, CA 94545
(800) 227-2776

GENERAL INSTRUMENTS
600 West John Street
Hicksville, NY 11802
(516) 733-3107

GTE ELECTRONICS
2000 West 14th Street
Tempe, AZ 85281
(602) 968-4431

HARDCORE COMPUTING
Box 44549
Tacoma, WA 98444
(206) 531-1684

HAYDEN SOFTWARE
50 Essex Street
Rochelle Park, NJ 07662
(800) 343-1218

HOWARD W. SAMS & CO., INC.
4300 West 62nd Street
Indianapolis, IN 46206
(800) 428-3696

INCIDER
80 Pine Street
Peterborough, NH 03458
(603) 924-9471

INFOWORLD
530 Lytton Avenue
Palo Alto, CA 94301
(415) 665-1330

INTERNATIONAL APPLE CORE
908 George Street
Santa Clara, CA 95050
(408) 727-7652

DON LANCASTER
Box 809
Thatcher, AZ 85552
(602) 428-4073

LAZER SYSTEMS
925 Loma Street
Corona, CA 91720
(714) 735-1041

LJK ENTERPRISES
Box 10827
St. Louis, MO 63129
(314) 846-6124

DAVID W. MEYER
600 Columbus Street
Salt Lake City, UT 84103
(801) 359-2790

MICROCOMPUTING
80 Pine Street
Peterborough, NH 03458
(603) 924-9471

MICRO INK
34 Chelmsford Street
Chelmsford, MA 01824
(617) 256-3649

MICRO LOGIC CORP.
Box 174
Hackensack, NJ 07602
(201) 342-6518

MICRO SCI
17742 Irvine Blvd.
Tustin, CA 92680
(714) 731-9461

MICROSOFT
10700 Northrup Way
Bellevue, WA 98004
(206) 828-8080

MICRO SPARC
10 Lewis Street
Lincoln, MA 01773
(617) 259-9039

MITEL
360G Leggett Drive
Kanata, Ontario K2K 1X5
(613) 592-5630

MOS TECHNOLOGY
950 Rittenhouse Road
Norristown, PA 19401
(215) 666-7950

MOTOROLA SEMICONDUCTOR
Box 20912
Phoenix, AZ 85018
(602) 244-6900

NEC ELECTRONICS
532G Broadhollow Road
Mellville, NY 11747
(213) 973-2071

NCR MICROELECTRONICS
1635 Aeroplaza Drive
Colorado Springs, CO 80916
(303) 596-5795

NIBBLE
Box 325
Lincoln, MA 01773
(617) 259-9710

PEELINGS
Box 188
Las Cruces, NM 88004
(505) 526-8364

QUALITY SOFTWARE
6660 Reseda Blvd.
Reseda, CA 91355
(213) 344-6599

RAK-WARE
41 Ralph Road
West Orange, NJ 07052
(201) 325-1885

ROCKWELL INTERNATIONAL
3310 Miraloma Avenue
Anaheim, CA 92803
(800) 854-8099

SAN FRANCISCO APPLE CORE
1515 Sloat Blvd.
San Francisco, CA 94132
(415) 556-2324

S-C SOFTWARE
Box 280300
Dallas, TX 75228
(214) 324-2050

SIERRA ON-LINE
36575 Mudge Road
Coarsegold, CA 93614
(209) 683-6858

SOFTALK
11160 McCormick Street
North Hollywood, CA 91603
(213) 980-5074

SOUTHWESTERN DATA SYSTEMS
10761 Woodside Avenue
Santee, CA 92071
(619) 562-3221

STELLATION TWO
Box 2342
Santa Barbara, CA 93120
(805) 966-1140

SYNERGETICS
Box 1300
Thatcher, AZ 85552
(602) 428-4073

SYNERTEK
Box 552
Santa Clara, CA 95052
(408) 988-5600

TEXAS INSTRUMENTS
Box 401560
Dallas, TX 75240
(214) 995-6611

THUNDER SOFTWARE
Box 31501
Houston, TX 77231
(713) 728-5501

WASHINGTON APPLE PI
Box 34511
Bethesda, MD 20817
(202) 332-9012

WESTERN DESIGN CENTER
2166 East Brown Road
Mesa, AZ 85203
(602) 962-4545

APPENDIX C

LABEL LISTS TO COPY

LABEL LIST FOR ____________________

DONE BY ____________ ASSEMBLER ____________

DATE __ – __ – __ SYSTEM ____________

VERSION ____________ ____________

LABEL	EQU	LINE	DFB	VALUE	USE

NOTES ____________________

PAGE ☐ OF ☐

LABEL LIST FOR

DONE BY		ASSEMBLER	
DATE	– –	SYSTEM	
VERSION			

LABEL	EQU	LINE	DFB	VALUE	USE

NOTES ____________________

PAGE ☐ OF ☐

Index

Book begins as Part One at

http://www.tinaja.com/ebooks/aacb1.pdf

SYNERGETICS SP PRESS

3860 West First Street, Thatcher, AZ 85552 USA
(928) 428-4073 http://www.tinaja.com

ISBN: 978-1-882193-17-2

Assembly Cookbook for the Apple™ II/IIe

(part two)

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

- Tells you what an assembler is, discusses the popular assemblers available today, and details the essential tools for assembly language programming.
- Covers source code details such as lines, fields, labels, op codes, operands, structure, and comments-just what these are and how they are used.
- Shows you the "new way" to do your source code entry and editing and to instantly upgrade your editor/assembler into a super-powerful one.
- Shows you how to actually assemble source code into working object code. Checks into error messages and debugging techniques.
- Includes nine ready to go, open ripoff modules that show you examples of some of the really essential stuff involved in Apple programming. These modules will run on most any brand or version of Apple or Apple clone, and they can be easily adapted to your own uses.

This cookbook is for those who want to build up their assembly programming skills to a more challenging level and to learn to write profitable and truly great Apple II or lie machine language programs.

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

ISBN: 978-1-882193-17-2