The Sourceror's Apprentice

The Assembly Language Journal of Merlin Programmers Vol 1 No 12 January 1989

Play it Again, SApp...

Belay that last issue, matey.

Last month I told you all about my determination to move *The Apprentice* into the "next level" in the publishing hierarchy. What I didn't tell you is that I've been negotiating like a maniac for weeks trying to launch a new publication - which would consume *The Apprentice* and replace *Call A.P.P.L.E.*

Allow me to explain.

I think we all agree that the Apple II programming community desires a multi-language technical journal. *Call A.P.P.L.E.* filled that niche nicely for years. I do like *nibble* for some things, but a true technical journal does not publish RoidBlaster and the like. It strives to propogate detailed programming knowledge and up to date information. I'm not really being critical of *nibble*, mind you - I don't think they make any claims towards technical journalhood.

That means, then, that *Call A.P.P.L.E.'s* death leaves a void - which I am determined to fill. I think we at the ol' Ariel cabin can do it, and do it well, so I'm going to lay it all on the line and see what you think.

Enter *8/16*

The new publication will be called *8/16*. Like its predecessor, *8/16* will be heavy on the source code. The following environments will be covered in monthly columns: Orca C, Micol Advanced Basic GS, Merlin 8 and 16 bit assembly, Applesoft, and ZBasic. All other environments will be represented as we get articles and code. At present, we have spoken to authors interested in writing about Pascal and APW/Orca assembly. The magazine looks like it will be 48 pages per month, about 15% of which will be devoted to advertising. We plan to keep the type size small, the listings single column wherever possible, the colors black and white, and the clip art minimal. I am inclined to call it a "no-nonesense" kind of programming journal. A one-year subscription will be $29.95.

Unlike *Call A.P.P.L.E.*, *8/16* will publish a monthly companion diskette (available for $69.95 for one year, $39.95 for six months, and $21 for three months). The disks will have approximately 300K of 8 bit and 300K of 16 bit code, articles, product demos, public domain software, and just about anything else you could imagine. All of the article text and source code for the current issue of *8/16* will be in there, as well as a few additional articles and tutorials that are not in the hardcopy version.

We shall fulfill existing *Apprentice* subscriptions (and *Reboot* and *Znews*) on an issue-for-issue and disk-for-disk basis. I think this is a good deal because you'll be "trading up", gettting a 48 page publication in place of a 12 pager. As a proud Merlin supporter, I assure you that Merlin assembly listings will always have a prominent place in the new journal. My goal is to provide you with what you have been getting and then throw in a whole lot more, too.

The key to making it all happen is going to be attracting advertisers. By combining our three Apple II publications, we have a built-in subscriber base right from the start. I have already created and sent out advertising kits to over 30 Apple II hardware and software companies. If you know of *anyone* who expresses the slightest bit of interest in advertising in *8/16*, please drop me their name and address - I'll ship 'em a kit ASAP.

I am very excited about *8/16*, and I'd like to thank Jack Nissel, Wally Matusak, Roger Wagner, Mike Westerfield, Jay Jennings, and Eric Mueller for their encouragement and support. You folks helped bring this to pass.

Speaking of Eric Mueller...

Eric Mueller, a frequent contributor to *The Apprentice*, has joined our staff as an Associate Editor. He'll be in charge of the GS sections of *8/16*, both in the magazine and on the disk.

As a favor to me while we "retool" for the new publication, Eric is producing this month's issue of *The Sourceror's Apprentice.*

For your information, Jerry "The Ampersand King" Kindall (currently the editor of *Reboot*) is going to be the editor of the 8 bit sections of *8/16*.

NOTE: Missing a Month

The first issue of *8/16* will hit the newsstands and your mailboxes on March 1st, 1990. To make this happen, **there will be NO February issue** of *The Apprentice*. If you have not received the third *and* fourth SApp quarterly disks (for those who ordered it), please drop us a note or give us a call. We want to have our books up to date and accurate by the time we make The Transition.

I've not come out and said it yet, but this is obviously the very last issue of *The Sourceror's Apprentice.* I am glad that, for once, the disappearance of an Apple II publication is not the final act of a financial tragedy. Still, it is the end of an era, albeit a short one. I hope that you end up feeling that your support of us this year has been rewarded by the greater range and depth of the new *8/16.*

Certainly, only time will tell, but I pledge our best efforts.

== Ross ==

From the Generic Mohawk Department:

Trapping Tricky Tool Errors

By Jay Jennings

We've seen more than enough Generic StartUp routines to last for quite a while. In order to continue our "Generic" articles, I had to come up with a routine that's useful and...generic. This error trapping routine is used in all my programs. In fact, it's stuck right in my generic startup source code file so I never forget about it.

Most error trapping routines that check for toolbox errors send your program straight to the bottomless pit. They're usually called with a piece of code like this:

```
          ldx       #$1111                ;our error ID number
          jsr       Check4Error
```

And the routine itself looks like this:

```
Check4Error bcs     :error                ;if carry set we got an error
          rts                             ;otherwise get outta here
:error
          pha                             ;save the toolbox error code
          stx       :ErrCode              ;save our error ID number
          ~HexIt    :ErrCode              ;change it to a hex number
          PullLong  :ErrCode              ;get and save the result
          ~SysFailMgr #:DeathMessage      ;display our death text

:DeathMessage dfb   #:endMessage-:startMessage
:startMessage asc   'At $'
:ErrCode    ds      4
            asc     '; Error $'
:endMessage
```

Now, there's nothing wrong with this method, but when you're developing a program, you're liable to crash your program more often than not. And rebooting every 10 seconds after crashing into Sliding Apple Hell gets to be a major hassle. With the help of Lane Roath and Eric Mueller, the Generic Error Trapper has evolved into a k-rad subroutine that's very useful and handy.

There are two major changes in the error routine. Instead of using an error ID number to show where our program died, we can use a complete string of characters. And instead of calling SysFailDeath, you get the chance to continue with the program or to jump straight to your shutdown routine. This way, you can continue if you know the error that just occurred isn't necessarily fatal, and if you do decide to quit, you won't have to reboot the machine.

By using a macro we can call the error routine like this:

```
          ~NewHandle #1024;ProgID;#$8000;#0
          CheckError 'NewHandle call in INIT.S file'
```

The error code itself is pretty straightforward. We use an AlertWindow to allow the program to continue execution, or branch to our ShutDown routine. And we take advantage of the substitution string capabilities of AlertWindow to print our error string and toolbox error code. (Check your back issues of SApp for more information on using this powerful feature of AlertWindow.)

First of all, let's look at the macro that makes this nifty routine so quick and easy to use.

```
CheckError mac
          ldx       #string              ;get low address
          ldy       #^string             ;get high address
          jsr       Check4Error          ;do that error check thing
          bra       ]here                ;if we came back, no error
string    asc       ]1                   ;our error string text
          dfb       00                   ;end of text mark (C string)
]here     eom                            ;end of the macro
```

This macro is very simple. It grabs the address of our error string and puts it in the X and Y registers. Then, we call the actual error trapping routine. If there was no error in the last tool call, we'll come right back and branch around the error string that's in our code. However, if there was an error, the address of the error string will be used to print it in our AlertWindow.

Here's the code for our new and improved error checking routine.

```
Check4Error ent
          bcs       :error               ;if there was an error, branch
          rts                            ;otherwise, get outta here
:error
                                         ;new code (below) will be added HERE
          stx       :SHR1                ;save the low word of the address
          sty       :SHR1+2              ;save the high word of the address
          sta       :ErrCode             ;save the tool error
          ~HexIt    :ErrCode             ;change code to a hex number
          PullLong  :ErrCode
          ~AlertWindow #0;#:Array;#:Text
          pla                            ;grab the result of AlertWindow call
          beq       :rts                 ;if 0 (Continue), we should continue
          jmp       ShutDown             ;otherwise, shutdown NOW!
:rts
          rts

:Text     asc       '62|Fatal error *0: *1. You should shut down.|'
          asc       'Continue|^Shut Down',00
:Array    adrl      :ErrCode
:SHR1 adrl -1
:ErrCode  asc       '    ',00            ;four spaces
```

There you go...a generic toolbox error checking routine. Unfortunately, this only works for desktop based programs. Fortunately, adding some code to make it work as well under the text environment is relatively easy.

By adding the following code you'll have an error checking routine that prints an error message on the text screen or the SHR screen.

Since there's no way I know of to tell whether the SHR screen is active or not, we can do a bit of juggling and get a routine that should work in most cases. To the beginning of the error routine (right after the `:error` label), we add some code to see if the Window Manager has been started, like this:

```
        phx
        phy                     ;save address to error string
        pha                     ;save the error code
        ~WindStatus             ;see if the Window Mgr. is up
        pla
        bne     :DoSHR          ;if WM is started, use AlertWindow
        brl     DoText          ;otherwise, use text stuff
:DoSHR
        _GrafOn                 ;make sure SHR screen is turned on
        pla                     ;restore our registers
        ply
        plx
```

If the Window Manager is active, we can be pretty sure that the tools we need for the AlertWindow are started up as well, and we can go on with the routine as normal. The call to `GrafOn` is there just in case the program was switched into text mode when the error happened. There are not many programs that switch from one mode to the other, but several of mine do, so it's better to be safe than sorry.

If the Window Manager is not active, we'll use a modified version of `TLTextMountVolume` to put the message on the screen. It's modified only to the extent that the message we use doesn't say to put in a specific disk (which is what it's normally used for). Here's the code that will place our error message and toolbox error code on the text screen:

```
DoText
        _GrafOff                ;make sure text screen is showing
        pla                     ;restore our registers
        ply
        plx
        stx     50
        sty     52
        sta     :TErrCode+2     ;save our error number

        sep     #$30            ;go down to <gulp!> 8-bits
        ldy     #0
]loop   lda     [50],y          ;get a character
        beq     :GotIt          ;if zero, we're at the end
        sta     :Text1+1,y      ;save it for our use
        iny
        cpy     #36             ;have we done 36 chars yet?
        bne     ]loop           ;if not, keep looping
:GotIt
        sty     :Text1          ;save the length byte
        rep     #$30            ;back to 16-bit everything

        ~HexIt  :TErrCode+2     ;change code to a hex number
        PullLong :TErrCode+2

        ~TLTextMountVol #:Text1;#:TErrCode;#:QuitBttn;#:ContBttn
        pla                     ;see what button they chose
        cmp     #2              ;2 = ESC, 1 = CR
        beq     :rts
        jmp     ShutDown        ;go to our shutdown routine
```

```
:rts
          rts
:Text1    ds        40            ;space for the error string
:TErrCode str       '$    '       ;dollar sign & four spaces
:ContBttn str       'Continue <ESC>'
:QuitBttn str       'ShutDown <CR>'
```

Notice that, since `AlertWindow` uses C strings (NULL terminated) and `TLTextMountVolume` uses Pascal strings (with a leading length byte), we have to transfer the error message into a Pascal string. There is another way around this, as `AlertWindow` will also deal with Pascal strings if you tell it to do so. Since all my other `AlertWindow` calls use C strings, I decided to remain consistent and juggle the string a little if the text version of the error window is required.

Also, the maximum length of the error string is 36 characters in the text version. This is because `TLTextMountVolume` switches to a 40 column screen (for some unknown reason!) to display it's message.

There are a couple of potential problems with this routine as written. Since we turn on or off the SHR screen (depending on what mode we need for display purposes), you may not be in the correct mode if you decide to continue instead of shutting down. So far that hasn't actually been a problem for me, but I can see where you might want to modify the code so this couldn't happen.

From the Mailbag 'o Fun Department:

Mike Mulls the Month's Mail

By Mike Rochip

Hi Mike...

I finally got a chance last night to sit down and read the November/December issue of SApp. I wanted to congratulate you on the slightly different format...I think that the "modular" programming stuff will be very useful.

I did want to point out a couple of descrepancies in this issue, though:

1. In the Demo Module, you declare your externals this way:

EXT Imprint,errorlist,MLI_Error

Since I use Merlin 16, I won't have any problems, but Merlin 8 users must put each external on a separate line. You might want to make a note of it for new users...they may become confused when they can't get it to work.

2. In the article itself, you say "Try to not to get excited when the demo tells you your volume bitmap may be damaged". Well, you won't have to worry about that happening...the demo can never cycle through MLI error $58. In line 49 of the demo, and in line 34 of the MLI error handler, you load X with a #28. There are, however, 30 errors in the table. It should have been loaded with a #29. Since the error message for the "volume bitmap damage" is in upper/lower case, and the other are in all caps, it appears that you might've added the "volume bitmap damage" message at the last minute, and forgot to increment the X register to allow for it.

3. Also, because of the demo module itself, you will never see error $1. To see error $1, the X offset would need to be loaded with "0". In line 62 (of the demo module), you only branch on a BNE.

As soon as the X register hits "0", the program will end, instead of doing the $1 error msg. Change the BNE to a BPL, and it will work...the branch won't take place until X becomes $FF on the next loop.

Sorry to cause so much trouble from my first read-through of SApp, but I thought that you'd want to know. I'm not trying to be picky...I think that SApp is a GREAT programming magazine, and I LOVE IT! Keep up the good work!

Tom Hoover
Lorena, Texas

Dear Mike:

Eric Soldan's article in the September issue is fine, but I think there is a bug in the source code, on page five. As published, the beginning of PDLADD reads:

```
PDLADD      plp       ;restore interrupts
            adc #0    ;add value of carry
```

Since we cleared the carry before saving the interrupt status (with a clc, php several lines up), the adc #0 line does exactly nothing. I think the right code for this section must be:

```
PDLADD          adc #0      ;add value of carry
                plp         ;restores interrupts
                            ;and clears carry
```

Nobody is perfect!

Sincerely,

Yvan Koenig
France

From the Parentheses and Brackets Department:

Steve Stephenson Goes To The Bank(s)

By Steve Stephenson

One of the prices we pay as assembly language programmers is the responsibility of managing memory. (I believe that high level languages were intended for those who would not accept this responsibility... but I'm not here to pursue that discussion today.)

In my last letter, I extolled the virtues of using a mini-direct-page in the stack and the Direct Page Indirect Long Indexed address mode (LDA [0],Y). Well, as most of you know (or should know), that addressing mode may only be used when you are sure that the block of memory you're addressing is all on the same bank. When you ask the Memory Manager for a block of memory to use, the attributes must include 'attrNoCross' ($xx1x) to guarantee that the block is completely contained in the same 64k bank.

But what do you do when you'd like a block larger than 64k? Or simply want to ease the demands on the Memory Manager and make more efficient use of all the odd scraps of memory? Well, for openers, you don't set 'attrNoCross' when you ask for the block. You also must use a different addressing mode! Or, as sure as bugs, when you least expect it, the Memory Manager will grant you a block that straddles two banks! The Direct Page Indirect Long Indexed addressing mode wraps around back to address $0000 on the lower bank instead of continuing into the higher bank (very much like the JMP ($20FF) bug in the 6502). This creates a bug of the worst order: the kind that doesn't always appear, and is practically impossible to duplicate.

So what addressing mode can you use for these situations? Well, you can always use the Absolute Long Indexed mode (LDA >$2000,X or LDAL $2000,X). But that only works if you know ahead of time where the block of memory is—which is not good style for a desktop program on the IIgs. The addressing modes that most nearly match the Direct Page Indirect Long Indexed while allowing for bank crossing are the Direct Page Indirect Indexed (that's the good old LDA ($0),Y we've been using for years) and the really scary looking Stack Relative Indirect Indexed (LDA (1,S),Y).

Actually, both of these alternatives are nearly identical. They both add the value of Y to the base address, which means they can reach a full 64k. They both form the base address from a two byte pointer that is located in bank zero. More importantly, they both transparently increment into the next bank when required. It's just what's inside the parentheses that's different; one uses a pointer in the direct page and the other uses a pointer in the stack.

There's a big catch to using these modes to reach anywhere in memory, though: you have to reset the Data Bank Register. And restore it afterward. It gets a little messy when you realize that your program is on one bank and the memory block is (most likely) on another bank. But I haven't found a better way to access a runtime-assigned block that could cross bank boundaries.

Some code fragments may help at this point. For this example, we'll assume that we are trying to put some known variable stuff (mystuff & morestuff) from our own code area into a block of memory that could be anywhere. Getting the new handle, locking it, and dereferencing the handle have already been done. The address of the block (myblock) is stored in your code space. And we are in full native (mx %00) mode.

```
*-------------------------
* the Direct Page style...

        lda  myblock+2      ;get the block's bank
        xba                 ;flip it around
        pha                 ;put it on the stack (A is 16 bits)
        plb                 ;but PLB pulls only 8 bits; throw it away (zero)
        plb                 ;  now Data Bank = memory block bank
        ldx  myblock
        stx  zptr           ;put block's address into direct page
        lda  >mystuff       ;must use long addr. to reach 'back' to our code bank
        sta  (zptr),y       ; else it would try to read 'mystuff' from this bank!
        iny
        iny
        lda  >morestuff
        sta  (zptr),y

        phk
        plb                 ;reset Data Bank back to our code bank

*-------------------------
* the Stack Relative style...

        lda  myblock+2      ;get the block's bank
        xba                 ;flip it around
        pha                 ;put all 16 bits on stack
        plb                 ;pop first 8 bits and throw away (zero)
        plb                 ;now Data Bank = memory block bank

        ldx  myblock
        phx                 ;put block's address on top of stack
        lda  >mystuff       ;must use long addressing to go back to code bank
        sta  (1,s),y
        iny
        iny
        lda  >morestuff
        sta  (1,s),y

        plx                 ;fix the stack (pop the block's address)

        phk
        plb                 ;and reset Data Bank back to our code bank
```

These methods may be harder to follow, but if you need to float your block of memory anywhere, then they may be your only answer. The stack relative method may be the only method if you have no direct page.

The biggest problem I've had using these methods is forgetting to use long addressing when the data bank is temporarily reset.

So, if your memory is all on the same bank, feel free to use the new square brackets... but if it isn't, then you need to be using good old parentheses.

Screen snatchin'

By Ross Lambert and Eric Mueller

At the heart of almost any application using the popular desktop metaphor is a screen pick routine (also known a screen snatcher). This useful (and powerful) chunk of code is designed to quickly save a piece of the screen and later, put it back up in the display memory. (For example, it's the key to the menu manager's menu caching.)

This month, we're presenting a double hi-res screen snatch routine: with it, you can pass top, left, bottom, and right coordinates, and the routine will lift your data right off the screen into a buffer. At that point, you can destroy the screen (within the rectangle you just saved, of course). To fix the screen back up, it's simply a matter of telling the snatch routine where your data was stored, and that you want it to be put back on the screen.

In order to cut down on code size and make it more applicable to desktop applications, the program's coordinate system mirrors the 80 column screen — the smallest piece of data that the snatch routine can work with is exactly the size of one 80 column screen character, seven pixels wide by eight pixels tall. If you're working with a double hi-res character generator, this screen snatcher is perfect! In order to keep within screen bounds, the largest X coordinate you may pass is 79 and the largest Y coordinate you may pass is 23.

Because of the usefulness of this routine, it was decided to make the program completely relocatable. As a result, it can be loaded anywhere in memory and may be used from Applesoft, ZBASIC, and just about any other language you can call it from.

Let's dive right into the code. Once we're past the header information and miscellaneous assembler pseudo-ops (lines 1-22), we start with the equates. Most are clearly commented, but let me mention a few important ones: line 26, "data", must be filled in before you call this routine. It's a two byte pointer to the buffer where you wish the screen data to be stored. In order to determine how many bytes to reserve for the buffer, use this formula: data_buffer_size = ((right - left) * 8) * (bottom - top), or, put differently, data_buffer_size = (horiz_width * 8) * vert_height.

Lines 35-38 are the X and Y coordinates for the top left corner and the bottom right corner of the rectangle you wish to save. Remember, set these coordinates as if you were dealing with the 80 column screen... 0-79 for the horizontal axis and 0-23 for the vertical axis.

Finally, line 39, "direction", is a very important flag: it controls whether or not you want to move data from the screen to the buffer (save the image) or from the buffer to the screen (restore the image). Poke a zero for the former and a one for the latter.

The program starts at line 53 by locking out interrupts, calling a known RTS in the monitor ROM, allowing interrupts to begin again, and then leaping over the lookup table. This lookup table (lines 58-81) is used to determine the base address for each of the 24 vertical positions we can address on the DHR screen.

At "start1" (line 87), the program checks on the stack for the last return address, placed there by the JSR to the known RTS, above. Once we have the address, we store it in the zero page pointer PTR, and then, with PTR, self-modify the two location-dependent lines of code in the program: the instructions at labels "modify1" (line 119) and "modify2" (line 122).

Once we have those locations properly set, and a couple of miscellaneous variables initialized (lines 112-113), we're ready to start moving the screen data about! "Yloop" (line 115) is the outer loop for the entire program. It starts by locating the base address for the line (0-23) that we're going to store/restore from. The inner loop, "Xloop" (line 126), begins next.

After setting the pointer "BASE" to the correct base address, the program determines if it should work with main or auxiliary memory and set the softswitches and an internal flag ("aux_flag") appropriately (lines 125-135). At the label "main", the program adds the horizontal offset to the base address in order to get a pointer to the actual character cell we're interested in saving.

Here, at lines 147-148, the program branches depending on what you wish to do: if you want to store data into your buffer, we continue on through to the label "byteloop" at line 153. However, if you chose to restore data to the screen from your buffer, the program branches to the label "restore" at line 202. Let's take a look at "byteloop" first.

Once determining what page we're working with (main or auxiliary) and flipping the appropriate softswitch, we loop through the storage of one byte, at lines 158-167: get a byte from the screen, flip to main memory, get the offset into the data buffer, put the data byte away, and bump the offset forward. If we're not done with the entire eight-byte-tall single character block (the check is at line 169), then we continue to loop through "byteloop".

When we are finished storing that character cell, the routine starting at line 181 ("Xck") takes over. It first checks to see if we're done with this row (horizontally), and if not, branches back to "Xloop". If we are finished with that row, we fall to "Yck" (line 188) to see if we're done with all of the rows (vertically). If not, the code returns to "Yloop" for the next row down. Otherwise, it falls through to "exit" and return to the calling routine.

The "restore" code (starting at line 202) is exactly the same as the store code, except that the process is reversed. In other words, from lines 202-219, we get the offset into the buffer, get a byte of the data, set the appropriate softswitches depending on the screen column, and then drop the byte onto the screen. Once an entire screen cell is completed, the routine finishes (line 222) by going back up to "Xck".

```
1                  lst      off
2       ***************************************
3       *                                     *
4       * DHR Screen Snatch                   *
5       * by Ross W. Lambert                  *
6       * Copyright (C) 1988                  *
7       * All Rights Reserved                 *
8       *                                     *
9       * version 2.1s (with 'Eric M0dz')     *
10      * January 13, 1990                    *
11      *                                     *
12      ***************************************
13
14                 xc                 ;allow 65c02 opcodes
15                 mx       %11       ;eight bit everything
16                 dsk      dhr.save
17                 exp      off       ;don't expand macros
18
19      orgAddr    =        $8000     ;put this code here
20                                    ; (it is relocatable - change to anything)
21
22                 org      orgAddr
23
24      * zero page & stack
25      BASE       =        $06       ;our zero page pointer
26      data       =        $08       ;FILL IN: pointer to screen data storage
27      PTR        =        $0A       ;current position of this program's data
28
29      STACK      =        $100      ;address of stack
30
31      * Page 3 freespace
32      st_offset  =        $0300     ;data offset storage
33      oldbase    =        $0301     ;old base address for each screen byte
34      aux_flag   =        $0303     ;auxilliary memory flag
35      xtop       =        $0304     ;FILL IN: x coord of top left
36      ytop       =        $0305     ;FILL IN: y coord of top left
37      xbot       =        $0306     ;FILL IN: x coord of bottom right
38      ybot       =        $0307     ;FILL IN: y coord of botom right
39      direction  =        $0308     ;FILL IN: 0=save screen/1=restore screen
40
41      * ROM
42      return     =        $FF58     ;harmless RTS in ROM
43      STOR80off  =        $C000     ;softswitch to make PAGE2on point to main mem p2
44      STOR80on   =        $C001     ;softswitch to make PAGE2on point to aux.mem p1
45      STOR80rd   =        $C018     ;reads status of 80STORE toggle switch
46
47      PAGE2off   =        $C054     ;selects page 1 main mem always
48      PAGE2on    =        $C055     ;selects p1 aux mem if STOR80on selected
49      PAGE2rd    =        $C01C     ;reads status of PAGE2 switch
50
51      ************************************
52
53      start      SEI                ;lock out interrupts
54                 JSR      return    ;goto harmless RTS to leave our LOC-1 on stack
```

```
            CLI
            BRA     start1          ;jump over table

screen      DW      #$2000          ;line 0      DHR LOOKUP TABLE
            DW      #$2080          ;line 1
            DW      #$2100          ;line 2
            DW      #$2180          ;line 3
            DW      #$2200          ;line 4
            DW      #$2280          ;line 5
            DW      #$2300          ;line 6
            DW      #$2380          ;line 7
            DW      #$2028          ;line 8
            DW      #$20A8          ;line 9
            DW      #$2128          ;line 10
            DW      #$21A8          ;line 11
            DW      #$2228          ;line 12
            DW      #$22A8          ;line 13
            DW      #$2328          ;line 14
            DW      #$23A8          ;line 15
            DW      #$2050          ;line 16
            DW      #$20D0          ;line 17
            DW      #$2150          ;line 18
            DW      #$21D0          ;line 19
            DW      #$2250          ;line 20
            DW      #$22D0          ;line 21
            DW      #$2350          ;line 22
            DW      #$23D0          ;line 23

 ****************************************************************************
 *  figure out where the heck we are and store on zero page
 ****************************************************************************

start1      TSX                     ;put stack pointer into X
            DEX
            CLC
            LDA     STACK,X         ;get low byte of our current position
            ADC     #5
            STA     PTR
            INX
            LDA     STACK,X         ;get high byte
            ADC     #0
            STA     PTR+1           ;and store it

            LDY     #modify1-orgAddr-7 ;modify location of screen lookup table
            LDA     PTR
            STA     (PTR),Y
            INY
            LDA     PTR+1
            STA     (PTR),Y

            LDY     #modify2-orgAddr-7
            LDA     PTR
            STA     (PTR),Y
            INY
            LDA     PTR+1
            STA     (PTR),Y

            STZ     st_offset       ;clear storage offset
            LDA     ytop            ;get vertical position  (YTOP)

Yloop       PHA                     ;store on stack
            ASL                     ;double it (lookup table in 2 byte pairs)
            TAX                     ;shift to use as offset

modify1     LDA     screen,X        ;get lowbyte of base address from lookup table
            STA     oldbase         ;store it
            INX
modify2     LDA     screen,X        ;now do highbyte of base address
            STA     oldbase+1
```

TSX
CLC
LDA !STACK-1,X
ADC #5
STA PTR
INY
LDA STACK,X
garage loader

lda #0
STA st. offset

code loaders

```

         LDA     xtop
Xloop    PHA                     ;store current horizontal position on stack
         LDX     oldbase         ;get back old base address for this vert. line
         STX     BASE            ;store it onto zero page
         LDX     oldbase+1       ;now do highbyte
         STX     BASE+1

         STZ     aux_flag        ;clear flag
         LSR                     ;divide by two (due to divided nature of DHR screen)
         BCS     main            ;cols 0,2,4,6, etc in aux mem/ 1,3,5 in main
         STX     aux_flag        ;set auxmem flag

main     CLC                     ;add horizontal position/2  to base address
         ADC     BASE
         STA     BASE
         LDA     BASE+1
         ADC     #$00
         STA     BASE+1

         LDX     #7              ;init byte counter
         LDY     #0              ; and offset

         LDA     direction       ;snatch screen (0) or restore it (1)?
         BNE     restore

**************************************************
* reads in all 8 bytes in one screen block

byteloop LDA     aux_flag        ;are we in auxmem?
         BEQ     main2           ;0 means main memory
         SEI                     ;lock out interrupts
         STA     PAGE2on         ;set softswitch

main2    LDA     (BASE),Y        ;get screen data from 1st byte in block
         STA     PAGE2off        ;set softswitch so we're back in main mem
         CLI
         LDY     st_offset       ;get storage offset
         STA     (data),Y        ;store it

         INY
         BNE     save_os         ;did it roll over to zero?
         INC     data+1          ; - if so, increment highbyte of data ( <>$FF )
save_os  STY     st_offset       ;save new offset
         LDY     #0              ;clear y offset
         CPX     #0
         BEQ     Xck             ;all done with screen block
                      Done C = 1
         DEX                     ;else decrement counter
         CLC   [illegible]
         LDA     BASE+1     ADC #3
         ADC     #$04            ;add 1024 to get next byte in screen block
         STA     BASE+1     CLC
         BRA     byteloop   BCC

***********************************

Xck      PLA                     ;pull off current horizontal position
         CMP     xbot            ;compare current hpos to ending hpos
         BCS     Yck             ;if done, advance Y loop
                  ADC #1
         INC      CLC BCC        ;increment horizontal position
         BRA     Xloop

Yck      PLA                     ;get last vertical line done off stack
         CMP     ybot            ;compare current vertpos to ending vertpos
         BCS     exit            ;if done, exit

         INC                     ;inc vertical line #
```

```
               BRA      Yloop             ;and branch back

exit           RTS


*************************************************************************
* Restore indicated portion of screen
*************************************************************************

restore        LDY      st_offset         ;get current data storage offset
               LDA      (data),Y          ;get data
               INY                        ;incrememt data offset
               BNE      :3                ;did it roll over?
               INC      data+1            ;if so, increment highbyte
:3             STY      st_offset         ;save new offset

               LDY      #0                ;clear offset
               PHA                        ;store data for awhile
               LDA      aux_flag          ;aux mem?
               BEQ      main3             ;if not, jump right to main
               SEI                        ;lock out interrupts when in aux mem
               STA      PAGE2on

main3          PLA                        ;get back data
               STA      (BASE),Y          ;store byte to screen memory
               STA      PAGE2off          ;repoint us to main memory
               CLI

               CPX      #0                ;all done with screen block?
               BEQ      Xck
               DEX      [illegible]       ;if not, dec counter and add next offset
               CLC
               LDA      BASE+1            ;only need to add to highbyte
               ADC      #$04              ;add 1024
               STA      BASE+1            ;can't ever wrap
               BRA      restore
```

CLC
BNE branche toujours.

The Sourceror's Apprentice

Copyright 1989 by Ross W. Lambert and Ariel Publishing
All Rigts Reserved.

Editor:	Eric Mueller
Publisher:	Ross Lambert
Subscriptions:	Tamara Lambert
Technical Editors:	Jay Jennings, Jerry Kindall
Stamp Licking:	Rebecca Lambert

Apple, Apple II, IIgs, BASIC.SYSTEM, and ProDOS are registered trademarks of Apple Computer, Inc.

We here at Ariel Publishing freely admit our shortcomings, but nevertheless strive to bring glory to the Lord Jesus Christ.

Editorial correspondence:

6547 N. Academy Blvd. Apt. #443
Colorado Springs, CO 80918
(719) 531-5893 GEnie: E.MUELLER

Subscription-related correspondence:

Box 398
Pateros, WA 98846
(509) 923-2249 GEnie: R.W.LAMBERT

Subscription prices in US dollars (Canada and Mexico, add $5, non-North American orders, add $18 per year): one year...$29.95, two years...$56

Back issues are $3 each (non-USA, add $2). There is a quarterly source code diskette available for $25 per year (Canada and Mexico add $5, non-North American orders add $15).

WARRANTY AND LIMITATION OF LIABILITY: I warrant that the information inThe Apprentice is correct and somewhat useful to somebody somewhere. Any subscriber may ask for a full refund of their last subscription payment at any time. At no time shall I or my contributors be held liable for any incidental or consequential damages in excess of the fees paid by a subscriber. **And don't forget...**Apple II Forever!