The Sourceror's Apprentice

The Assembly Language Journal of Merlin Programmers Vol. 1 No. 6 June, 1989

PEEKing in AuxMem, Switching Banks, and a Couple Bad Jokes

I hope you enjoyed last month's 24 page extravaganza. It was very enjoyable (though a tad expensive) to be able to bring you such a thick issue. Don Lancaster calls such a deal "personal value added" - i.e. giving the customer a smidgin more than they expected and in a manner only you can deliver. I hope to be able to do stuff like that often. BTW, Don's book, *The Incredible Secret Money Machine,* is a fun and totally hip guide to starting your own business. There is a lot of sage advice within its pages. And it appeals to the aging hippie within me, too.

Speaking of business, I need to remind those of you who have charged your subscriptions that our name will appear as "Teacher's Software Co." on your bill. Please don't do a "charge back"; it is really us. We changed our name with the bank long ago, but the VISA and MASTERCARD folks are the epitome of unresponsiveness. Now that we're back in civilization I'm changing banks - that ought to get it straightened out (don't anyone ask if I'm in the main bank or the aux bank, though ... booo, I know).

Some of my "friends" have been making fun of my sense of humor lately. That should teach 'em. Neener neener neener.

Back Issues?

...are three dollars each, including shipping and handling. We started in January with Volume 1, Number 1. And yes, we've been slower than break-up in Unalakleet in getting some of them out to you. However, by the time you read this everyone should have their back orders (assuming you ordered back issues before June 20th). If we've messed you over, our apologies, and please let us know. We'll make it right ASAP.

And yes, I agree with those of you who would like a commented listing of the articles in the back issues. I'll include a one page insert every now and again (next month?).

The Quarterly Disk?

...will be out within 10 days or so after you receive this newsletter. That makes a little sense if you stop to consider that it includes the source code and most of the text for this issue.

A couple people have asked about the "SAPP DISPLAYER" code and some of the funny binary files on the disk (like RT.AUX, etc.) I hope no one shoots me over this, but SAPP DISPLAYER is a SFGETFILE work-alike written in ZBASIC™. The funny binary files are part of the ZBASIC run-time package. I meant to put that on the title screen and forgot (Oops, sorry Zedcor!). BTW, I am chasing down a screwy little bug that seems to mess up my mouse routines on a IIc. If you had problems with your SAPP disk in that regard, it is our problem, not yours. I hope to have it fixed post haste.

Incidentally, we publish ***Znews***, too, a ZBASIC programming newsletter. And if you'll forgive a tiny commercial: we're giving away ZBASIC at our cost ($42) and we'll throw in a sample ***Znews*** - this because we believe you'll love the language (we're hooked) and you'll subscribe to the newsletter ($29.95 for 1 year).

Where's Mike?

Poor Mike Rochip, he's been left out in the cold again this month. My source code cup brimmeth over, so Mike and I both got elbowed out into next month's issue.

On tap this month: I finally have an opportunity to print the 16 bit version of Steven Lepisto's "Vectored Joystick

Programming" code (the article and the first part of the 8-bit code ran in March. The rest of the 8-bit code ran in April).

If you make use of his subroutine notice that the responsiveness (i.e. the cycles used) changes depending on your screen position. If you need a constant number of cycles per read you'll want to insert some sort of scaling factor. I'll see what I can do about that myself in a future issue.

This month's feature presentation is another 8-bit adventure (See? I haven't been neglecting you 8-bit afficionados! It so happens that most of my contract work is 8 bit.) Matt Neuberg's "AUXPEEK" program is a utility designed to allow us to probe auxiliary memory in very much the same manner that the monitor probes main memory. I wish I had this program months ago when I was working on some double high resolution graphics routines. It would have saved some headaches, I think.

There is a lot I like about his code, particularly the clever manner in which he returned program control to main memory. I guess I ought to let you see for yourself, though. Dig in and enjoy!

PEEKing at Auxiliary Memory: a Monitor Utility

by Matthew Neuberg

The Monitor's Blind Spot

Anyone using the Monitor to snoop around inside a 128K machine (enhanced IIe, IIc, or Laser 128), has probably encountered an annoying limitation: the Monitor is incapable of reporting on the contents of auxiliary memory. In effect we have a 128K computer of which the Monitor can see only 64K.

If, however, a program which we are trying to develop or investigate uses or partly lives in auxiliary memory, the ability to peek into the 2nd 64K can be crucial. Since the Monitor cannot do this for us, we will write a utility of our own: AUXPEEK. The exercise will not only result in a valuable tool for investigation and development, but will also teach us something about program interaction across the main/auxiliary RAM boundary.

128K Memory Architecture

Even though all 128K of a 128K Apple (or Laser) may in theory be accessed immediately by a program as it is running, or may be interpreted as a program and executed by the computer's microprocessor, the fact is that the microprocessor can only think at any given moment about addresses within a range of 64K, because the Program Counter is only 16 bits. Therefore the 128K is divided into two 64K groups, called Main and Aux RAM, only one of which can be an object of the microprocessor's attention at any given moment.

But the situation is in reality more complicated than this. Each 64K of RAM is itself divided into groups. Memory in the range of addresses $200-$BFFF is treated as a unit, called the 48K memory. On the other hand, the zero-page, the page 1 stack, and the addresses $D000-$FFFF are treated as a separate unit, called bank-switched memory. This name derives from the fact that the range of addresses $D000-$DFFF actually refers to two 4K groups of memory, called Bank 1 and Bank 2, though once again the computer cannot think about both banks simultaneously. (Thus e.g. the address $D000 can refer to any of four data bytes: main bank 1, main bank 2, auxiliary bank 1, or auxiliary bank 2.)

Note: Firmware memory, occupying the region $C000-$CFFF, is not treated in this article. Some computers have more than one bank of memory in this region (""expansion ROM"), containing important routines, but AUXPEEK will not permit us to examine these.

The purpose of this division into units is to enable the units to be switched between Main and Aux separately. There are three distinct sections of memory and three questions we must examine, all of which are answered by setting softswitches, namely : 1) Which 48K bank should the CPU address? 2) Which bank-switched memory (Main or Aux) should the CPU address, and 3) Within the Main or Aux banks of question 2, which 4K bank should the CPU address? A possible setting might thus be: Main 48K RAM ($200-$BFFF), Aux bank-switched RAM (zero-page, page 1 stack, and $D000-$FFFF), and Bank 1 ($D000-$DFFF).

But the soft-switches do not simply select which regions of memory the computer is to think about absolutely. Rather, the concept of ""thinking about" is divided into two sorts of operation: reading and writing. For example, LDA is a ""read" operation; STA is a ""write" operation. It is important to understand what our options are in this connection.

In the case of 48K memory, the soft-switches allow us to set separately the memory group (Main or Aux) to which each sort of operation is to apply. A program running entirely within Main 48K memory has the switches set so as both to read and to write in Main 48K memory; but we can in fact set the switches, say, so as to read from Main 48K memory and write to Aux 48K memory. Under such a setting, for example, a sequence of commands `LDA $2000, STA $2000` would transfer the contents of Main $2000 to Aux $2000. (The Accumulator, to and from which our LDA-ing and STA-ing is performed, is unique and not a part of memory, which is why it can be used as an intermediary in this transfer.)

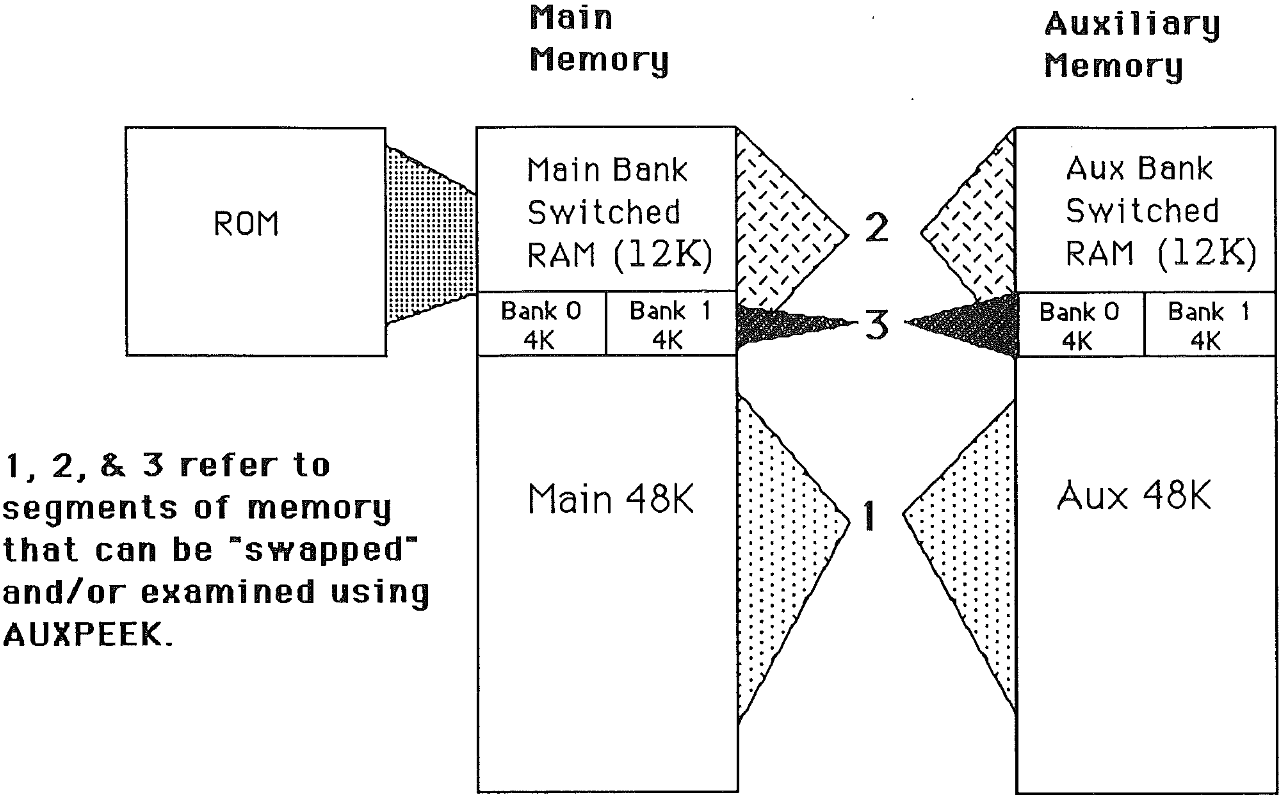
In the case of bank-switched memory, the situation is different. First of all, when we select Main or Aux, we must commit ourselves as part of that selection to using Bank 1 or Bank 2 of the region $D000-$DFFF. Secondly, we cannot select one of Main or Aux for reading and the other for writing, and, even when we select Main or Aux $D000-$FFFF for reading only, we have automatically selected the corresponding (Main or Aux) zero-page and page 1 stack for both reading and writing.

The Problem

The above facts are important because they have dictated the way in which AUXPEEK operates. First, let's decide on grounds of convenience to have AUXPEEK live in Main page 3, where it is least likely to interfere with anything. Moreover, we may as well give AUXPEEK maximum value by enabling it to peek at any part of memory outside Main 48K (that is, not only Aux 48K, but also Main or Aux bank-switched memory, and either Bank 1 or Bank 2).

Now, it will be very easy to peek at any bank-switched memory address: we have only to select the desired bank and Main or Aux bank-switched memory (which will not affect our program in page 3, since page 3 is not part of bank-switched memory), and then LDA directly from the desired address.

Figure 1. - A Simplified 128K Apple II Memory Map



On the other hand, we will have to be very clever in order to read from Aux 48K memory. It is easy for a program running in Main 48K to poke a value into Aux 48K; we have seen above how to do this. But a program running in Main 48K can by no means of itself peek at (read from) Aux 48K. This is because, in a Von Neumann machine (and all modern computers are Von Neumann machines), programs live in memory as, and are indistinguishable from, data. This means that if a program running in Main 48K throws the switch commanding the microprocessor to perform subsequent read operations from Aux 48K, the microprocessor, having upped the Program Counter appropriately, will look to Aux 48K for the next program instruction, because fetching a program instruction counts as a read operation. But if our program lives entirely in Main 48K, it won't find it, and we are heading for a crash.

Solutions

Most programs which use 128K transfer information between Aux and Main memory through the use of built-in firmware routines. If the address of data to be transferred is known absolutely, it is possible to use the routine AuxMove (or MoveAux), which copies a block of memory from Main 48K to Aux 48K or vice versa. This approach, however, lacks flexibility: the manual warns us that it works only within 48K memory, and besides, to prepare a call to AuxMove requires considerable program space, something of which we are particularly jealous, since we are confining AUXPEEK to page 3.

A more complex solution is to place into Aux memory, in advance, a routine for obtaining data from Aux memory, and then, at the appropriate moment, transfer control to that routine via the computer's firmware routine XFer. (This, for example, is the method used by Glen Bredon's SOURCEROR.) This method is especially useful when we must use indirect addressing to obtain our data: the address of the desired information is stored in the zero-page; then control is passed to the routine in Aux memory, which does an indirect-addressing read from Aux memory and then transfers control back to the appropriate place in Main memory, carrying the desired value in the accumulator. But this method has the same drawback as using AuxMove, and besides, it also requires the overwriting of much Aux memory, so that we might overwrite something we wanted to peek at.

Our solution is to exploit to the fullest the nature of the problem itself. In our main program, we will go ahead and throw the switch commanding subsequent read operations to come from Aux memory; but we will in advance have planted some code at the corresponding next program address in Aux memory __ where the processor will then find and execute it. This code will simply perform a direct LDA, and then throw the switch commanding read operations to come from Main memory once again, thus transferring control back to our main program.

This solution has two great advantages. First, we will still have to overwrite some of Aux memory, but only 6 bytes. Second, we will copy our code into Aux memory from within our program in Main memory at exactly the same addresses; this means that, just in case we are called upon to peek, not at Aux memory, but at Main (bank-switched) memory, we can bypass the command to read from Aux memory, and the LDA command will be in place right in our program in Main memory.

But how will the LDA command planted in Aux memory know what address to read from? We do not want to use indirect addressing, because this will involve modifying the zero-page and add other complications. The simplest solution is to have our program modify itself. As soon as we know what address we want to read from, we will copy that address into our program code right after the LDA code. Then when we copy our two lines of code into Aux memory, the LDA command will already be correct.

(Note: Occasionally one sees a claim that self-modifying code is bad programming practice. My response is that if you don't like self-modifying code you've no business either using a Von Neumann machine or writing machine code: this sort of technique is just what they're for!)

Other Implementation Details

The Monitor includes a user-command facility: the command CTRL-Y, which simply causes a call to $3F8. We will therefore include in AUXPEEK a header which puts at $3F8 a JMP to our main routine and then does an RTS; this header will be run only when AUXPEEK is first loaded, via a BRUN command. Moreover, since the program memory occupied by the header is then superfluous, it will subsequently be used during calls to AUXPEEK for data storage.

Obviously AUXPEEK must be able to parse a keyboard command. Since this would be extremely consumptive of program space, we will have the Monitor do the parsing for us via the routine GETNUM ($FFA7), which, though not a ""legal" entry point, is reliable for both 128K Apples and the Laser.

GETNUM expects a Monitor command in the input buffer, starting at $200,Y. A Monitor command consists of up to 4 hex digits, followed optionally by a (non-hex) upper case letter. GETNUM halts when it encounters either a (non-hex) letter or a CR. After GETNUM halts, A2 ($3E/$3F) contains the numeric part of the command; the Accumulator contains the item that caused the halt, that is, either a letter if one was encountered or a CR if not; and Y indexes the item within the input buffer after the item that caused the halt.

(Editor: During the course of editing this article, I discovered that Professor Neuberg's assumptions in the paragraph above were 100% accurate - for the Laser, his machine of choice. On an Apple II, the accumulator holds an encoded value after returning from GETNUM (as opposed to a true ASCII code). For this reason I inserted a `DEY` ***, an*** `LDA $0200,Y` ***, and a quick*** `INY`***. This loads A with the value of the non-hex character that bumped us out of GETNUM and properly restores the Y offset, thereby creating the conditions the good professor needed for the rest of his code to work. I highlighted my changes with boldfaced comments.)***

To make life easier, we will have AUXPEEK show us, each time it is called, 8 bytes starting at the byte named in the command. We won't make any attempt to arrange these bytes in mod-8 groups, as the Monitor does. We will, however, imitate the Monitor to this extent: immediately after one call to AUXPEEK, following the display of 8 bytes, a subsequent command CTRL-Y, with no address attached, will be sufficient to cause the display of the next 8 bytes. This will be valuable in case we want to look at a sizeable block of bytes. Every time AUXPEEK prints a group of 8 bytes, it will precede it with the address of the first byte, printed in inverse, to distinguish it from output of the Monitor's own memory display routines.

Installation and Command Syntax

As stated above, installation consists of BRUNning AUXPEEK from BASIC. It will then be installed into page 3 memory, with the CTRL-Y vector pointing at it.

You can then enter the Monitor via CALL -151. For safety's sake, you should probably put yourself into 40-column mode (using ESC CTRL-Q) before issuing any commands to AUXPEEK.

Subsequent to installation, calls to AUXPEEK may be given to the Monitor, in response to the Monitor's asterisk-prompt. A legal command consists of CTRL-Y (which will not appear on the screen, alas) followed immediately, as part of the same line and without spaces, by up to 4 hex digits denoting the starting address to be peeked at. (A CR, of course, terminates the command.)

For example, the command [CTRL-Y]A100 will cause the display of 8 bytes starting at Aux $A100. Moreover, we will have AUXPEEK select by default the Aux bank-switched RAM, so commands for the Aux zero-page and stack will have the same syntax, e.g. [CTRL-Y].

On the other hand, if the address to be examined lies in the range $E000-$FFFF, we will give the user the option of specifying either Main or Aux bank-switched RAM, by the addition of the letter M (Main) or X (auX) to the command. Thus [CTRL-Y]F000M will cause the display of 8 bytes starting at $F000 of Main bank-switched RAM. And, if the address to be examined lies within the 4K range $D000-$DFFF, the user should specify, in addition to Main or Aux, Bank 1 or 2, by the addition of 1 or 2 to the command: e.g. [CTRL-Y]D100M1 shows 8 bytes starting at $D100 of Bank 1 of Main bank-switched RAM.

Finally, as stated above, the command [CTRL-Y], if given directly after another AUXPEEK command, will cause the display of the next 8 bytes.

AUXPEEK In Detail

When a command line is gathered by the Monitor, using GETLNZ, it is placed into the input buffer at $200. The Monitor then calls GETNUM to parse the command. As soon as the first character, CTRL-Y, is encountered, GETNUM halts, and the Monitor passes control to AUXPEEK.

AUXPEEK sets Y to 1, just to be on the safe side, so that in the upcoming call to GETNUM the CTRL-Y at $200 will not be encountered. GETNUM is then called, and it parses the contents of the input buffer, starting at $201, loading up to 4 hex digits of the command into A2.

Now if, after calling GETNUM, the accumulator holds a CR, we know that the command consisted at most of hex digits. Moreover, if the accumulator holds a CR and Y is 3, then the command must have consisted of just 2 bytes, namely, CTRL-Y and a CR, and no hex address. Finally, if the accumulator does not hold a CR, then it holds either M or X (unless the command is illegal), and Y indexes either a 1 or a 2, or something we can ignore (such as a CR or something illegal). If an illegal item is encountered, we jump back into the monitor with a beep.

A variable MAINAUX is maintained, with the options for bank-switched memory X-or-M, 2-or-1 encoded in bits 6 and 7 respectively, where they can be easily checked by a BIT operation. If the user command is just [CTRL-Y], MAINAUX is left unchanged from the last time AUXPEEK was called, and the address to be read is fetched from within the program, where it was stored after that last call. Otherwise, MAINAUX is zeroed, which is interpreted as the default option Aux Bank 2, and then if the user's command consists of more than just hex digits, bits are rolled into MAINAUX to set it appropriately.

We then print in inverse the first address to be shown, also transferring that address into the LDA command within the program.

Next we copy the LDA command, and the following read-from-Main-48K command, into Aux memory. It is then a simple matter to use MAINAUX to select the correct soft-switches and read a byte of data. The byte is stored in the workspace that used to be occupied by the header. We then increment the address within the LDA command. We loop so as to perform the operations described in this paragraph 8 times; when we are done, the 8 bytes of data are in our workspace.

The 8 bytes of data stored in the workspace are now printed, and we are done, so we jump back into the Monitor with no beep. The Monitor prints a prompt and waits for the next command.

Voila!

```
* AUXPEEK
* a control-Y monitor utility to display AUX memory
*  and bank-switched RAM
* Matt Neuburg -- 3/20/89

```

```
           =003E  A2       EQU   $3E        ;and $3F, set by GETNUM
           =0200  IN       EQU   $200       ;the input buffer
                  * monitor routines
           =FE80  SETINV   EQU   $FE80      ;print in inverse
           =FE84  SETNORM  EQU   $FE84      ;print normal
           =FDED  COUT     EQU   $FDED      ;print char in acc
           =FDDA  PRBYTE   EQU   $FDDA      ;print byte in acc
           =F941  PRTAX    EQU   $F941      ;print bytes in acc, X
           =FFA7  GETNUM   EQU   $FFA7      ;parse monitor command
           =FF65  MON      EQU   $FF65      ;print *, await monitor
command
                           ORG   $300
                  * header to initialise ^Y-vector, & variable storage
000300: A9 4C     TEMP     LDA   #$4C       ;initialise ^Y vector:
"JMP"...
000302: 8D F8 03           STA   $3F8
000305: A9 13              LDA   #<START    ; ...to main routine
000307: 8D F9 03           STA   $3F9
00030A: A9 03              LDA   #>START
00030C: 8D FA 03           STA   $3FA
00030F: 60        MAINAUX  RTS
                  *-------------------------------
000310: 4C 65 FF  NOGOOD   JMP   MON        ;err, return to Monitor with
Beep
                  *-------------------------------
                  * start of ^Y routine: parse command
000313: A0 01     START    LDY   #1         ;ignore the CTRL-Y
000315: 20 A7 FF           JSR   GETNUM     ;stick addr into A2
000318: 88                 DEY             ;redundant for Laser but
000319: B9 00 02           LDA   IN,Y      ;necessary on A2 - RWL
00031C: C8                 INY             ;bump Y back
00031D: C9 8D              CMP   #$8D       ;is that all there is?
00031F: D0 19 =033A        BNE   CHEKSYN    ;=>no
000321: C0 03              CPY   #3         ;yes, is it just ^Y CR?
000323: B0 0D =0332        BGE   DEFAULT    ;=>no
000325: AD A2 03           LDA   READ1+1    ;yes, don't alter MAINAUX,
000328: 85 3E              STA   A2         ; and set A2 ourselves
00032A: AD A3 03           LDA   READ1+2    ; using last inc'd value
00032D: 85 3F              STA   A2+1
00032F: 4C 5B 03           JMP   PRINT      ;==> done parsing
000332: A9 00     DEFAULT  LDA   #0         ;command was just addr:
000334: 8D 0F 03           STA   MAINAUX    ; so zero MAINAUX (= aux bank
2)
000337: 4C 5B 03           JMP   PRINT      ;==> done parsing
00033A: C9 CD     CHEKSYN  CMP   #"M"       ;there's more (M, M1, etc.):
00033C: F0 04 =0342        BEQ   VALIDMX    ; so check syntax, must be M
or X
```

```
00033E: C9 D8                  CMP   #"X"
000340: D0 CE =0310            BNE   NOGOOD
000342: 4A           VALIDMX   LSR                ;this bit is auX=0, Main=1
000343: 6E 0F 03               ROR   MAINAUX      ;put in bit 7 (will be 6
later)
000346: B9 00 02               LDA   IN,Y         ;(Y has been incd by GETNUM)
000349: C9 B1                  CMP   #"1"         ;continue checking syntax,
00034B: F0 0A =0357            BEQ   VALID12      ; must either be 1 or 2...
00034D: C9 B2                  CMP   #"2"
00034F: F0 06 =0357            BEQ   VALID12
000351: A5 3F                  LDA   A2+1         ;or else addr must >= $E000
000353: C9 E0                  CMP   #$E0         ; (and in that case, won't
matter
000355: 90 B9 =0310            BLT   NOGOOD       ; what "bank" bit we roll in)
000357: 4A           VALID12   LSR                ;this bit is 2=0, 1=1
000358: 6E 0F 03               ROR   MAINAUX      ;put in bit 7

                     * all roads lead here: print starting addr in inverse

00035B: 20 80 FE     PRINT     JSR   SETINV
00035E: A5 3F                  LDA   A2+1
000360: A6 3E                  LDX   A2
000362: 8D A3 03               STA   READ1+2      ;also, copy starting addr...
000365: 8E A2 03               STX   READ1+1      ; into prog for direct LDA
later
000368: 20 41 F9               JSR   PRTAX
00036B: 20 84 FE               JSR   SETNORM
00036E: A9 BA                  LDA   #":"
000370: 20 ED FD               JSR   COUT

000373: A0 07                  LDY   #7           ;initialise for indexing TEMP
                                                  ; and to loop what follows 8
times

                     * copy the LDA addr command into AUX mem

000375: A2 05        XFER      LDX   #XEND-READ1-1 ;index bytes to copy
000377: 8D 05 C0               STA   $C005        ;write to AUX mem
00037A: BD A1 03     SEND1     LDA   READ1,X      ;copy one byte from MAIN to
AUX
00037D: 9D A1 03               STA   READ1,X
000380: CA                     DEX                ;another?
000381: 10 F7 =037A            BPL   SEND1        ;=>yes, loop

000383: 8D 04 C0               STA   $C004        ;done, restore write to MAIN

000386: 2C 0F 03               BIT   MAINAUX      ;will we read from MAIN or
AUX?
000389: 8D 08 C0               STA   $C008        ;select MAIN zp and bank-RAM
00038C: 70 03 =0391            BVS   PICKBANK     ;...or...
00038E: 8D 09 C0               STA   $C009        ;select AUX zp and bank-RAM

000391: AD 88 C0     PICKBANK  LDA   $C088        ;select bank 1 bank-RAM read
000394: 2C 0F 03               BIT   MAINAUX
000397: 30 03 =039C            BMI   PICKRAM      ;...or...
000399: AD 80 C0               LDA   $C080        ;select bank 2 bank-RAM read

00039C: 70 03 =03A1  PICKRAM   BVS   READ1        ;if MAIN, =>do nothing
00039E: 8D 03 C0               STA   $C003        ;if AUX, read from AUX ram
```

```
                 * these are the two lines to be copied into AUX

0003A1: AD FF FF   READ1     LDA   $FFFF      ;"FFFF" modified by program
0003A4: 8D 02 C0             STA   $C002      ;restore read from MAIN ram

0003A7: 99 00 03   XEND      STA   TEMP,Y     ;now we have obtained a byte

0003AA: EE A2 03             INC   READ1+1    ;always inc addr
0003AD: D0 03 =03B2          BNE   NOTHI
0003AF: EE A3 03             INC   READ1+2
0003B2: 88         NOTHI     DEY              ;dec TEMP index and count
0003B3: 10 C0 =0375          BPL   XFER       ;do all that 8 times

                 * restore everything, print 8 bytes collected

0003B5: 8D 08 C0             STA   $C008      ;restore MAIN zp
0003B8: AD 81 C0             LDA   $C081      ;restore read ROM, write RAM 2
0003BB: AD 81 C0             LDA   $C081      ;(again)

0003BE: A0 07                LDY   #7         ;8 bytes to index and print

0003C0: B9 00 03   SHOWBYTE LDA    TEMP,Y     ;obtain a byte
0003C3: 20 DA FD             JSR   PRBYTE     ;and print it
0003C6: A9 A0                LDA   #" "
0003C8: 20 ED FD             JSR   COUT
0003CB: 88                   DEY              ;dec TEMP index and count
0003CC: 10 F2 =03C0          BPL   SHOWBYTE   ;loop 8 times

0003CE: 4C 69 FF             JMP   MON+4      ;back to Monitor, no beep

End Merlin-16 assembly, 209 bytes, errors: 0 , symbol table: $1800-$1911
```

Vectored Joystick Programming
IIGS Version

by Steven Lepisto

(Editor: Steven's article appeared in the March issue and the bulk of the 8-bit code ran in the April issue. This version is GS specific.)

```
         lst   off
         rel
         dsk   joystick16.1

         xc
         xc
         mx    %00

```

```
* Requires the following labels external to this file (preferably direct page):
*               (These should be word values)
* trigger     = - if button is down
* button_state = state of button(s).
*                0 = no button pressed        2 = button up
*                1 = button down              3 = button still down
* joyvectx    = direction of x coordinate: -1, 0, +1.
* joyvecty    = direction of y coordinate: -1, 0, +1.
*
* These variables are defined here arbitrarily so the file can be assembled.
* See documentation on ways to deal with these variables.

trigger  ent
         ds    2
button_state ent
         ds    2
joyvectx ent
         ds    2
joyvecty ent
         ds    2


* Macros used by these routines.

* SHORT and LONG use the following conventions:
* SHORT         : sets 8-bit A and X,Y regs.
* SHORT a_reg   : sets 8-bit A.reg
* SHORT x_reg   : sets 8-bit X,Y regs (actually, anything
*                  that doesn't start with 'a' will work).

SHORT    mac
         do    ]0
         if    a=]1
         sep   #%00100000
         if    mx&%01
         mx    %11
         else
         mx    %10
         fin
         else
         sep   #%00010000
         if    mx&%10
         mx    %11
         else
         mx    %01
         fin
         fin
         else
         sep   #%00110000
         mx    %11
         fin
         <<<
LONG     mac
         do    ]0
         if    a=]1
         rep   #%00100000
         if    mx&%01
         mx    %01
         else
         mx    %00
         fin
         else
         rep   #%00010000
         if    mx&%10
         mx    %10
         else
         mx    %00
         fin
         fin
         else
         rep   #%00110000
         mx    %00
         fin
         <<<
```

```
*------------------------------------------------
* Joystick read routine (16 bit version) for Apple IIgs.
* by Stephen P. Lepisto
* Date: 1/3/88
* Assembler: Merlin 16 v3.50+.
*
* Reads a standard analog joystick in a custom way.  Returns values that
* are 0-127 which is useful for vector-type motion.  Also reads the buttons
* and sets a global variable accordingly.  Combines both buttons into one.
*
* Note that these routines assume that there will be one call to dojoystick
* for every call to updatejoystick.  Updatejoystick adds to the state of the
* stick until dojoystick clears it so you can call updatejoystick more than
* once before you call dojoystick.
*
* Dokeystick: returns 0 if no joystick equivalent keys are pressed else
* returns a byte that looks like stickstate (see updatejoystick for
* specifics).
*
* NOTE: GS-specific in locations and in CPU instructions!
*
*------------------------------------------------
*
* To use these routines:
* 1) call initjoystick to intialize the routines and determine if there is a
*    joystick present.  Only has to be done once.
* 2) at top of main loop, call updatejoystick to get current state of stick.
* 3) sometime after calling updatejoystick, call dojoystick to process state
*   of stick and return vector and trigger values.
*
* If stick isn't present, updatejoystick and dojoystick only process button
* presses (a la the apple keys).  If you wish, you can allow for installing a
* joystick on the fly by having the user press a key then based on that key,
* call initjoystick again.  If the stick is ever unplugged on the fly,
* updatejoystick and dojoystick will fall back to reading only buttons,
* leaving the stick itself in a centered state.
*------------------------------------------------

* Hardware locations.

keypress equ    $e0c000     ;- if valid key press present
keystrobe equu  $e0c010     ;access to clear keypress
gs_speed equ    $e0C036     ;speed register of IIgs
resetstick equ  $e0c070     ;reset paddle timers
rdstickx equ    $e0c064     ;timer for paddle 0 (+ when done)
rdsticky equ    $e0c065     ;timer for paddle 1 (+ when done)
button0  equ    $e0c061     ; - if button 0 pushed
button1  equ    $e0c062     ; - if button 1 pushed

*------------------------------------------------

* Variables.

stick_last ds  1            ;last state of stick
stick_live ds  1            ;positive if it's really there
stick_temp ds  1
stickstate ds  1

*------------------------------------------------

```

```
        lda   stick_live
        and   #$ff
        xba
        rts



* Read keyboard looking for joystick equivalent keys.
*
* Output:
*   zero flag : set if no keypress processed or recognized else clear.
*   A.reg     : if zero flag set, holds a 0 else holds stick state byte.
*
* Note that only if a key is recognized is the keyboard strobe cleared.  This
* allows another routine outside of this one to see if the keypress was meant
* for it.
*
* Currently supports eight motions, a fire button, and a combination fire
* and motion button (to show how it can be done).
* Also supports P for pause (waits for another keypress), and ctrl-J for
* reinitializing the joystick (if it has been reconnected after first running
* the initialization routine).

        mx    %11
dokeystick
        ldal  keypress
        bpl   :x
        cmp   #"a"
        bcc   :0
        cmp   #"z"+1
        bcs   :0
        and   #$df
:0
        and   #$7f
        cmp   #'P'
        bne   :0a
        stal  keystrobe
:waitkey
        ldal  keypress
        bpl   :waitkey
        stal  keystrobe
        bra   :x
:0a
        cmp   #$0a       ;ctrl-J
        bne   :1
        jsr   initjoystick
        bra   :x
:1
        sta   dokey_char
        ldy   #-1
:2
        iny
        lda   key_table,y
        beq   :x
        cmp   dokey_char
        bne   :2
        lda   joyxlate_tbl,y
        stal  keystrobe
        rts
:x
        lda   #0
        rts

```

```
dokey_char ds  1

* Key equivalent table:
*
* Current order is:
* W : diagonal up left        X : down      F : button press
* R : diagonal up right       E : up        M : button press and down motion
* Z : diagonal down left      S : left
* C : diagonal down right     D : right

key_table dfb  'W','R','Z','C'
          dfb  'X','E','S','D'
          dfb  'F','M'
          dfb  0          ;end of table

* Values in this table correspond in position with the keys in key_table.

joyxlate_tbl dfb %00101,%01001,%00110,%01010
          dfb  %00010,%00001,%00100,%01000
          dfb  %10000,%10010

          mx   %00

* Processes last joystick read or current keyboard read (if any) and returns
* information about the joystick.
*
* Output:
*  joyvectx : -1, 0, +1 depending on x position of stick.
*  joyvecty : -1, 0, +1 depending on y position of stick.
*  trigger  : - if button event occured else +.
*  stickstate : before next updatejoystick, current state of stick.  Bit 4
*               reflects current position of button, set if button down.
*

dojoystick ent
          short
          jsr  dokeystick ;read and interpret keys as joystick
          bne  :1         ;branch if key equivalent pressed
          bpl  :a         ;button equivalent key not pressed
          lda  #0         ;else clear motion vectors
          sta  joyvectx
          sta  joyvectx+1
          sta  joyvecty
          sta  joyvecty+1
          bit  stick live
          brl  :6
:a
          lda  stickstate
:1
          sta  stickstate
          lda  stickstate

          cmp  stick_temp ;has the state changed?
          beq  :6         ;branch if not
          sta  stick_temp
          long x_reg
          ldx  #0         ;yes - which way?
          ldy  #0
          ror
          bcc  :2
          dey             ;up
:2
          ror
          bcc  :3
          iny             ;down
:3
          ror
          bcc  :4
          dex             ;left
:4
          ror
          bcc  :5
          inx             ;right
:5
          stx  joyvectx
          sty  joyvecty
          short x_reg
```

```
:6
        lda   stickstate
        asl
        asl
        eor   stickstate
        and   #%11000000
        beq   :nochange
        ldy   #2
        lda   stickstate
        and   #%00110000
        beq   :skipchange
        ldy   #1
        bra   :skipchange
:nochange
        ldy   #0
        lda   stickstate
        and   #%00110000
        beq   :skipchange
        ldy   #3
:skipchange
        sty   button_state
        stz   button_state+1
        tya
        lsr
        bcc   :8 ;button not down
:7
        lda   #-1
        bra   :9
:8
        lda   #0
:9
        sta   trigger
        sta   trigger+1
        lda   stickstate
        and   #%00110000
        sta   stickstate
        long
        rts


* Get values from joystick and convert to bit positions
* in 'stickstate'.
*
* The "dead space" around center is about 65%.
*
* Output:
* 'stickstate'
* bit 0 : = 1 if stick is up
* bit 1 : = 1 if stick is down
* bit 2 : = 1 if stick is left
* bit 3 : = 1 if stick is right
* bit 4 : = 1 if button pushed
* bit 4 : = 1 if button 0 pushed
* bit 5 : = 1 if button 1 pushed
* bit 6 : previous state of button 0
* bit 7 : previous state of button 1

updatejoystick ent
        short
        bit   stick_live
        bmi   :5
        jsr   readstick
        cpx   #255
        bne   :1
        lda   #-1
        bmi   :1a
:1
        lda   #0
:1a
        sta   stick_live
        lda   stickstate
        and   #%00110000
        asl
        asl
        bit   stick_live
        bmi   :5
        cpy   #16         ;up
        bcs   :2
        ora   #%00000001
:2
        cpy   #100        ;down
        bcc   :3
        ora   #%00000010
:3
        cpx   #16         ;left
        bcs   :4
        ora   #%00000100
:4
        cpx   #100        ;right
        bcc   :5
        ora   #%00001000
:5
        tax
        ldal  button0
        bpl   :6
        txa
        ora   #%00010000
        tax
:6
        ldal  button1
        bpl   :7
        txa
```

```
         ora   #%00100000
         tax
:7
         stx   stickstate
         long
         rts


* Read apple joystick, returning values for left/right, up/down directions.
*
* Output:
* x.reg = value for horizontal movement (0-127)
*       = 255 if no stick is attached.
* y.reg = value for vertical movement (0-127)
*
* Timing: minimum (both x,y read 0) = approx. 83 cycles
*   maximum (both read 127) approx. 3023 cycles
*   If no stick plugged in = approx. 5935 cycles

         mx    %11
readstick
         php
         sei
         ldal  gs_speed
         sta   oldspeed
         and   #$7f
         stal  gs_speed
         ldal  resetstick ;reset timers on all paddles
         ldx   #0
         ldy   #0
:1
         nop              ;delay tactics to compensate for
         nop              ;the inx/bne :2
         nop
:2
         ldal  rdsticky
         bpl   :4         ;branch if done reading
         iny
         beq   :5         ;escape hatch if stick not plugged in
         ldal  rdstickx
         bpl   :1         ;branch if done reading
:3
         inx
         bne   :2         ;always branches (it had better!)
:4
         nop
         nop
         nop              ;compensation for not doing the iny/beq :5
         ldal  rdstickx
         bmi   :3         ;branch if still reading
:5
         lda   oldspeed
         and   #$80
         oral  gs_speed
         stal  gs_speed
         plp
         rts

oldspeed ds    1

         mx    %00
```