The Sourceror's Apprentice

The Assembly Language Journal of Merlin Programmers Vol. 1 No. 3 March, 1989

News and Passion

Those of you who do ANY assembly language programming on the GS should be fairly quivering with excitement; Roger Wagner Publishing has just released *Merlin 16+*. I do not have the software yet, but I've been in touch with several of the beta testers. The most common remark I heard was, "...blazing speed." A friend of mine had to do a contract sort of job that was due "yesterday", and all he could say was, "Thank God for Merlin 16+".

I'll do a formal review for you after I've worked with it for awhile, but the brochure Roger Wagner sent me has some "almost too good to be true" features listed. I can see why it has taken a long time to develop this product. I'm sure Roger will send you a brochure if you haven't already received one. Probably the best feature in my mind is that current Merlin owners can get the new version for a whopping 65% off, $45.95 as opposed to $125. This seems to me to be a nice reward for current Merlin fans and supporters. Thanks, Roger. (RWP, 1050 Pioneer Way, Suite "P", El Cajon, CA 92020 619/442-0522)

You may be a little surprised to be getting your March issue of this newsletter so early (at least compared with the service of the last two months). We originally planned to publish *The Apprentice* in the middle of each month ('cuz that's the way things worked out initially), but Mr. Postman and Mr. CopyCenter have been getting progressively slower since last December. It seems bizarre to me that postal service was BETTER during the Christmas rush, but I think they geared up for Christmas with temporaries and then laid them off after the holidays. They must've ditched too many in Anchorage, because things have really been getting bottled up for me lately.

I don't like delivering a newsletter in the wrong month, so I moved things up. I cannot express how difficult it is to push a production schedule forward, but we've managed to do it, at least by a few weeks, anyway. We'll still be a little more erratic than *A+* (especially during our move at the end of May), but things are starting to get on more of an even keel.

I imagine that most of you are subscribers to Tom Weishaar's *A2-Central* (if you're not, you should be), but for those who are not, *A2-Central* is sponsoring an Apple II Developers Conference July 21-22 at Avila College in Kansas City. It looks like I'll be leading a session on ProDOS 8 programming environments which should be pretty interesting. The biggest news to me, though, is that Tom coerced the Beagle Bros. into disclosing the secrets of writing *TimeOut* applications. As soon as I heard that, I figured that I'd sell my car to get to this conference. No kidding. Y'all come now, ya heah? (See? I'm getting in the mood already.)

Speaking of Tom Weishaar, I was grateful for the plug in the March issue (thanks, Tom!); to be mentioned in the same sentence with Bob SanderCederlof and Uncle-DOS, even in jest, is probably more than I deserve at this point. So that there are no misunderstandings or hurt feelings (mine!), allow me to explain once more the origins of this newsletter.

I have done some professional work in assembly language on the Apple II, but I am not Bob S-C by any stretch of the imagination. Bob was (oops, he didn't die, I mean IS) a star even among the pros. I just couldn't stand to see *AAL* bite the dust.

I figured two things: First, if we sat around and waited for someone of Bob's caliber to start a

newsletter, we might have to wait a very long time. Technical expertise, an entrepenurial spirit, and gifts with the written word are tough characteristics to find in one person. A long wait for something like *The Apprentice* is not good for the II, me thinks. There is a need for an all-assembly language forum *right now.*

Second, I figured that my experience with newsletters put me in a fairly unique situation. I am sure that Tom Weishaar would agree with me when I say that the publishing business can be a tad tricky. There are a million ways to lose money and it is risky. In spite of Bob's prowess, *AAL* only had a little over 700 subscribers. I am already a full time publisher, and I have this publication on a steady business footing from day one.

Disassembling AppleWorks is beyond me, I think. It is certainly beyond the range of my desires!

My goals ARE to do the things I do well - that is, provide a regular forum in which the superstars can detail the cutting edge of technology (somebody talk Dave Lyons into writing for us), AND teach and explain things that are not regularly covered in the popular press. For example, even though *nibble* does print assembly language listings, the explanations and commentary are usually limited to what you need to do to get the software running (with the notable exception of the illustrious Sandy Mossberg). I plan an article detailing the workings of double high resolution graphics and a DHR DRAW commmand. It is not a state of the art topic, but my mail has told me that most folks are not state of the art. That is why I felt it important to spend so much time on the generic startup routine for the GS. Not only is it a little shy of straightforward, but part of the reason GS software acts a little flakey sometimes is that proper attention has not been paid to some of the 16 bit minutiae.

I hope I sound passionate and not defensive. Of all the projects I have undertaken, none has sparked my interest like *The Apprentice.* I believe it has the potential to enlighten the best of us with its right hand, and elevate the least of us with its left.

I invite your comments. Now back to our regularly scheduled program...

The Applesoft Connection: Part 2

by Jerry Kindall

In the first part of "The Applesoft Connection" I discussed how to pass numeric parameters to and from Applesoft variables. In this, the sequel, I want to show you one trick with numeric parameters, and then move on to strings.

Three-Byte Integers

One of the trickiest parts of writing MicroDot was figuring out how to deal with three-byte integers. I needed three-byte integers for the RANDOM module, which is a set of MicroDot commands for working with random-access data files. RANDOM has a command for moving ProDOS' file pointer to a specified byte in a file, and another command for reading the file pointer's position into an Applesoft variable.

Well, since ProDOS supports files up to sixteen megabytes in length, the file pointer is three bytes long. Everything I had read about Applesoft told me that only two-byte integers can be handled easily by Applesoft. It was time to dig into the Applesoft source code, provided with Merlin in the form of SOURCEROR.FP.

I found that with a little work, I could do what I wanted. The only drawback is that you can't return a three-byte integer to an Applesoft integer variable. You must use a real variable. Remember, a three-byte integer allows you to handle numbers from zero to 16,777,215 (sixteen megabytes).

Some of the Applesoft routines used in the program below were covered in the first part of "The Applesoft Connection." There are, however, a couple of new ones lurking about. They are FLO3 and QINT.

FLO3 ($EBA6) is actually an alternate entry point into the FLOAT routine at $EB93, which was covered in part 1. FLOAT has several entry points which are used by other routines in Applesoft. You might notice a distinct similarity between the way FLO3 is called and the way FLO2 (covered in part 1) is called (see lines 64-66).

QINT ($EBF2) is the Applesoft INT command. It converts a floating-point value in the FAC to an integer, also stored in the FAC. QINT (Quick INTeger) is also used by GETADR and COMBYT, covered in part 1.

Another new routine is TYPERR at $DD76. It's similar to IQERR and SYNERR covered in part 1. It issues a ?TYPE MISMATCH error.

```
 1 * The Applesoft Connection
 2 * Three-Byte Integers
 3 *
 4 * Routines by Jerry Kindall
 5
 6 * Zero page locations:
 7
 8 VARNAM    =    $81       ;variable name
 9 VARPNT    =    $83       ;variable pointer
10 FAC       =    $9D       ;floating-point accumulator
11
12 * Applesoft routines:
13
14 CHKNUM    =    $DD6A     ;check for numeric
15 TYPERR    =    $DD76     ;?TYPE MISMATCH
16 CHKCOM    =    $DEBE     ;check for a comma
17 PTRGET    =    $DFE3     ;get pointer to variable
18 IQUERR    =    $E199     ;?ILLEGAL QUANTITY
19 MOVMF     =    $EB2B     ;move FAC to variable
20 FLO3      =    $EBA6     ;convert integer to FAC
21 QINT      =    $EBF2     ;convert FAC to integer
22
23 * GETNUM3 routine
24 *
25 * Evaluates Applesoft expression and
26 * returns a three-byte integer in
27 * Acc, X-reg, Y-reg (lo/mid/hi)
28
29 GETNUM3   JSR  CHKCOM    ;we must have a comma
30           JSR  FRMNUM    ;evaluate numeric formula
31           JSR  QINT      ;get integer value
32
33           LDA  FAC+1     ;check highest byte (#4)
34           BEQ  :OK       ;if # < 16777215 it's OK
35           CMP  #$FF      ;negative equivalent?
36           BEQ  :OK
37           JMP  IQERR
38
39 :OK       LDA  FAC+4     ;get number: lo
40           LDX  FAC+3     ;            mid
41           LDY  FAC+2     ;            hi
42
43           RTS
```

```
44
45 * PUTNUM3 routine
46 *
47 * Pass a 3-byte integer back to an
48 * Applesoft real variable; value in
49 * Acc, X-reg, Y-reg (lo/mid/hi)
50
51 PUTNUM3   STA  FAC+3     ;store number: lo
52         STX  FAC+2     ;              mid
53         STY  FAC+1     ;              hi
54
55         LDA  #0
56         STA  FAC+4     ;zero fractional part
57
58         JSR  CHKCOM    ;comma check
59         JSR  PTRGET    ;find variable
60         JSR  CHKNUM    ;make sure it's numeric
61
62         LDA  VARNAM    ;must be real
63         BMI  :TM       ;reject integer
64
65         LDX  #$98      ;convert integer to FAC
66         SEC
67         JSR  FLO3
68
69         LDX  VARPNT    ;move value to variable
70         LDY  VARPNT+1
71         JMP  MOVMF
72
73 :TM       JMP  TYPERR    ;?TYPE MISMATCH
```

Three-byte integers may come in handy for other applications as well. For example, you could write a utility to POKE or PEEK any address in the Apple IIgs's memory range by using a combination of these routines and native-mode machine code.

Stringing Along

Numbers can be entertaining, but we've had our fun with them, so let's turn to strings. A lot of the concepts and routines we have covered work with strings just as well as they do with numbers, but there are definitely some new tricks to learn.

One important fact to remember is that Applesoft stores string variables in two parts. One is called the "descriptor" and is stored along with the variable's name in the variable table. The descriptor holds the length of the string and the address of the actual text contained by the string. The actual string text is stored in high memory, in a place called the string pool.

The first routine we need is called FRMEVL and lives at $DD7B. It's closely related to FRMNUM ($DD67), discussed in the first part of this series. FRMEVL evaluates an Applesoft formula and puts the result into the FAC. Unlike FRMNUM, though, FRMEVL will handle both numeric and string expressions. (FRMNUM actually calls FRMEVL and then jumps to CHKNUM to make sure the result is numeric.) As with FRMNUM, the formula evaluated by FRMEVL can be any legal Applesoft string expression.

After calling FRMEVL, we must make sure to call CHKSTR at $DD6C to make sure that the formula we just evaluated was a string. (As far as I know, there's no string equivalent of FRMNUM to evaluate an expression and check for a string result in one step.)

String Beans

To pass a string back to an Applesoft variable, we again use PTRGET to find the variable's address. Then we call CHKSTR to make sure that the variable is indeed a string. VARPNT will point to the string variable. **Remember, this isn't the actual string itself, it's the descriptor, which holds the** length of the string and the address of the string's actual contents. In short, when you're dealing with strings, VARPNT acts as a pointer to a pointer. GS programmers might recognize this use of VARPNT as an 8-bit precursor of the "handle."

The next step is to load the length of the string into the Accumulator and call STRINI (STRIng Init) at $E3D5. STRINI will collect garbage if necessary, get space for the new string in the string pool, and create a descriptor for the string in DSCTMP at $9D. (If you have been paying close attention you probably noticed right away that DSCTMP is at the same address as the FAC. When dealing with numbers, this location is called the FAC; when dealing with strings, it's called DSCTMP, for TeMPorary DeSCriptor.)

Then we load the X and Y registers with the current address of the string (low byte into X, high byte into Y). A call to MOVSTR ($E5E2) copies the string to its final resting place in the string pool.

All that remains is to copy the string's new descriptor to the variable. Remember that strings come in two parts; MOVSTR takes care of the text part, but we still need to take care of the descriptor.

String Routines

Here are some subroutines similar to ones I used in MicroDot. MicroDot has a "pathname buffer" at $280 which is used to hold a file's pathname so that it can be operated on. The MicroDot string routines are designed to move strings to and from this pathname buffer. Not only does the string need to be moved, it must be converted to the ProDOS format, which involves putting a length byte at the beginning of the string.

```
 1 * The Applesoft Connection
 2 * String Routines
 3 *
 4 * Routines by Jerry Kindall
 5
 6 * Zero page locations:
 7
 8 SPTR      =    $5E       ;pointer to string result
 9 DSCTMP    =    $9D       ;holds new string descriptor
10 VARPNT    =    $83       ;pointer to string variable
11
12 * Applesoft routines
13
14 ERROR     =    $D412     ;issue an error
15 CHKSTR    =    $DD6C     ;check for string
16 FRMEVL    =    $DD7B     ;Applesoft formula evaluator
17 CHKCOM    =    $DEBE     ;syntax check for comma
18 PTRGET    =    $DFE3     ;get pointer to variable
19 STRINI    =    $E3D5     ;init string space & pointer
20 MOVSTR    =    $E5E2     ;move string to string pool
21 FREFAC    =    $E600     ;free up temp string pointer
22
23 * Pathname buffer
24
25 PATH      =    $280      ;length byte followed by text
26
```

```
27 * GETPATH routine
28 *
29 * Evaluates string expression and
30 * places result into PATH
31
32 GETPATH    JSR  CHKCOM     ;skip past comma
33           JSR  FRMEVL     ;evaluate string expression
34           JSR  CHKSTR     ;make sure it's a string
35           JSR  FREFAC     ;free up the memory
36
37           CMP  #64        ;length must be < 64
38           BGE  :ERR
39           STA  PATH       ;set length byte
40
41           CMP  #0         ;check for null string
42           BEQ  :NULL
43
44           LDY  #0         ;copy string to PATH
45 :GET      LDA  (SPTR),Y
46           STA  PATH+1,Y   ;skip past length byte
47           INY
48           CPY  PATH       ;string length
49           BNE  :GET
50
51 :NULL     RTS
52
53 :ERR      LDX  #176       ;?STRING TOO LONG
54           JMP  ERROR
55
56 * PUTPATH routine
57 *
58 * Passes pathname in PATH back to
59 * an Applesoft string variable
60
61 PUTPATH    JSR  CHKCOM     ;skip past comma
62           JSR  PTRGET     ;get variable address
63           JSR  CHKSTR     ;make sure it's a string
64
65           LDA  PATH       ;get length
66           JSR  STRINI     ;get space & descriptor
67
68           LDX  #PATH+1    ;first byte is the length
69           LDY  #/PATH+1   ;so we really want PATH+1
70           JSR  MOVSTR     ;move it to string pool
71
72           LDY  #2         ;move descriptor to variable
73 :LOOP     LDA  DSCTMP,Y
74           STA  (VARPNT),Y
75           DEY
76           BPL  :LOOP
77
78           RTS
```

A Practical Routine

To round out our discussion of strings, here's yet another Input Anything routine, using the string-handling techniques described thus far. The string is read with the standard GETLN routine at $FD6A.

We then call GDBUFS ($D539), an Applesoft subroutine that masks off the high bits of all the characters in the input buffer. From there, we're back in familiar territory.

To use this routine in your Applesoft programs, use the statement CALL 768,A$. A$ will hold the entered string, including any commas, colons, and quotes, up to 255 characters worth. You can, of course, use any string variable in place of A$.

```
 1 * The Applesoft Connection
 2 * Yet Another Input Anything
 3 *
 4 * Routine by Jerry Kindall
 5
 6 * Zero page locations
 7
 8 DSCTMP     =    $9D        ;holds new string descriptor
 9 VARPNT     =    $83        ;pointer to string variable
10 PROMPT     =    $33        ;prompt printed by GETLN
11
12 * Input buffer
13
14 BUF        =    $200
15
16 * Applesoft & Monitor routines
17
18 GDBUFS     =    $D539      ;fix input buffer for BASIC
19 CHKSTR     =    $DD6C      ;check for string
20 CHKCOM     =    $DEBE      ;syntax check for comma
21 PTRGET     =    $DFE3      ;get pointer to variable
22 STRINI     =    $E3D5      ;init string space & pointer
23 MOVSTR     =    $E5E2      ;move string to string pool
24 GETLN      =    $FD6A      ;get an input line
25
26            ORG  $300
27
28 INPUTANY   JSR CHKCOM      ;check for comma
29            JSR PTRGET      ;find variable
30
31            LDA  #$80       ;no prompt
32            STA  PROMPT
33            JSR  GETLN      ;get input line
34
35            TXA             ;save string length
36            PHA
37
38            JSR  GDBUFS     ;make good Applesoft string
39
40            PLA             ;remember length
41            JSR STRINI      ;make space & descriptor
42
43            LDX  #BUF       ;get address of input buffer
44            LDY  #/BUF
45            JSR  MOVSTR     ;move it to string pool
46
47            LDY  #2         ;move descriptor to variable
48 :LOOP      LDA  DSCTMP,Y
49            STA  (VARPNT),Y
```

```
50              DEY
51              BPL   :LOOP
52
53              RTS
```

Coming Attractions

The next installment of The Applesoft Connection will tie everything we've looked at so far together, as we begin writing some real live ampersand routines. We'll take an in-depth look at the CALL statement and the ampersand statement and learn how to integrate machine language routines with both. See you then!

Custom Joystick: Vectored Joystick Programming

Stephen P. Lepisto
GEnie: S.LEPISTO
2130 S. Fairway
Pocatello, ID 83201

Custom Joystick is a joystick read routine that I've developed over the last two years to address the problem of reliably reading a joystick on the Apple II computer. I currently have two principle versions: one for all 8 bit Apples including the IIgs in emulation mode and one specifically for the IIgs. The listing talked about here is the 8 bit version. The coding differences between the two versions are not insignificant but both behave in the same way and accomplish the same task.

The original way of reading a joystick on the Apple II series computer was the use of a Monitor routine called PREAD located at $FB1E. You called it with the X register holding a number indicating which paddle you wanted to poll. It would return with a value in the A register from 0-255, depending on where the paddle was situated. A joystick on the Apple is essentially a combination of two paddles, one is for the x axis (left and right) and the other is for the y axis (up and down). So to deal with a joystick, you needed to call PREAD twice, once for each of the two axes on the stick. Well, that is the ideal situation.

In the real world, there is this quantity called time. It takes time to do anything. And this most definitely true when trying to read a joystick using PREAD. To put it simply, the timing circuits that are used to read a paddle take a finite amount of time to to do their thing. They also need to "breathe" between timing cycles (and allow a capacitor to discharge). Because of these timing constraints if you call PREAD once for one paddle then immediately call PREAD for the other paddle, you will most likely get erroneous (i.e., bad) results for the second paddle because the timing circuits may not have fully recovered from the first read. So one usual technique for using PREAD is to call PREAD for paddle 0, wait at least three milliseconds then call PREAD for paddle 1. Some programmers have even found it necessary to call PREAD twice for each paddle to get the right results. All of this calling and waiting takes time. Sure the whole process may not take more than 14 milliseconds on a 1 Mhz Apple but in machine language, you can do a whole heck of a lot in 14 milliseconds.

The advantage of using the PREAD approach is that you can get a value from 0-255 for each axis which gives you a large enough range to be useful on a graphics screen (i.e., if your range is large, you get better resolution of positioning over a large area). So if you were to hold the joystick in the center, then the cursor would be in the center of the window or screen (depending on how the program did that sort of thing). Move the stick to the left and the cursor would move a corresponding distance to the left until the stick was all the way to the left in which case the cursor would be all the way to the left of the window. This is called a position-oriented use of a joystick.

Now, on graphics screen where the resolution is a good deal larger than 256 (the range returned by

PREAD), position-oriented cursor moves are much more difficult because it becomes next to impossible to hit certain points on the screen. Enter the vector-oriented use of a joystick.

A vector-oriented joystick is one in which the the stick tells the program which direction to move the cursor. The program then moves that cursor at a constant rate until the joystick stops giving directions, usually by centering the stick. This gives you eight directions to move a cursor and the size of the screen is no longer a limitation. Since a vector-oriented joystick only needs enough information to distinguish between the eight directions and a center point, it is possible to use a shortcut in reading the joystick. That is where my routines come in.

My routines offer support for a vector-oriented use of a joystick which can accurately read a joystick much, much faster than using PREAD. This gives your program more time to do what it is supposed to be doing. As a bonus, my routines can even tell if a joystick is plugged in or not allowing automatic control over such things.

Let me be the first here to dispel any misconceptions about my routines being the only way to read a joystick. They're not. Many other programmers have found ways to read the stick and in many of those approaches can return values up to 512 or more. What I feel my stick read routine offers is speed and small space. I'm also offering my code in the public domain so more programs can take advantage of another peripheral input device, namely, the joystick.

The listing accompanying this article contains a set of six routines that are used for support of a vector-oriented use of a joystick. Two are necessary: Apple_ID and ReadStick. The other four, although useful in their own right, are not essential for reading a stick. Briefly, the routines are:

ReadStick : Reads the joystick in a custom way, accessing the hardware directly.

UpdateJoystick : Used to update the current state of the joystick (it calls ReadStick).

DoJoystick : Used to convert the results of UpdateJoystick to useful information. Keyboard equivalents are also handled at this point.

Dokeystick : Handles certain keys that are used as equivalent movements of the joystick. This way, if a joystick is supported, so is a set of keys that mimic the joystick.

InitJoystick : Initializes some variables needed by the other routines. It also determines if the joystick is present or not.

Apple_ID : Used to identify the type of Apple II computer these routines are being run on.

Specific details on each routine

What follows is a detailed description of each routine.

`ReadStick`

This is the heart of my joystick routines. Using a different method than PREAD to read the paddle circuits, I am able to poll two paddle circuits at the same time. This saves time and allows for quicker access to the joystick, giving better response.

The first thing I do upon entry is set the interrupt disable flag. Wouldn't do to be interrupted while reading the stick. Next, I call a routine that will slow the IIgs processor to 1 MHz if that is the computer the routines are being run in. It is my understanding the the IIc+ will automatically reduce its speed to 1 MHz for 5 milliseconds if any address in the range $C070-$C07F is accessed, so this case is automatically handled. Unfortunately, the Transwarp card was made with a part that has turned out not to be completely reliable so slowing it down can sometimes hang the system (which is not a good thing) so I don't support it. The two reasons I slow down the CPU is so I can maximize the range returned by the polling without adding to the complexity of the routine and so the routine behaves the same on all Apples.

Next thing that happens is I access $C070, which initializes the paddle timer circuits (Actually, any address in the range $C070 to $C07F will reset the paddle circuits but twiddling with anything other than $C070 may introduce possible conflicts with other hardware). Then I immediately enter the polling loop which checks each paddle timer in turn looking for an end to it all. As it polls each timer, it increments the appropriate index register. This is how the hardware can be converted into a number for the program's use. Note that there is an escape hatch after the INY. This is in case there is no stick present. If the escape hatch weren't there, the routine would be stuck in an infinite loop. As it is, we now have an easy way of determining the presence of a joystick.

In PREAD, the routine polls one paddle circuit, incrementing a single index register in the process. Because of this dedication, the range of numbers is 0-255. In my routines, I'm polling two paddle circuits at the same time so the range is a lot less.

When the polling is done, I reset the speed of the computer to its former state and restore the interrupt disable flag. Upon exiting the routine, the X register holds a value from 0-145 (representing the x axis on the stick) and the Y register holds a value from 0-145 (representing the y axis on the stick). In the case where there is no stick present, the X register will hold 255 and the Y register may hold 255 (the Y register has a slight priority over the X register in the routine so it will hit 0 as part of the escape hatch before the X register will. In any event, I have found that the X register is the one to always use to check for the presence of a stick).

`UpdateJoystick`

This routine is used to update the status of the joystick. You call it once per time through your main loop. If you call it more often than that you stand the chance of missing some button actions. The routine is designed so that it remembers the state of the buttons through two reads (so you get a previous state of the buttons and a current state) and it adds the current position of the stick to the remembered state. In addition, UpdateJoystick will determine the presence or absence of the stick and set a flag appropriately. Once the flag is set (meaning the stick isn't there), UpdateJoystick will only read the state of the buttons. There is a way to deal with installing a joystick on the fly which I will detail in DoKeyStick.

So, how does UpdateJoystick do all this wonderful stuff? First of all, if there is a stick present, it calls ReadStick to get the current position of the stick. Then it determines if there is a stick present or not and sets a flag indicating that status. If a stick is determined to be present, the values returned by ReadStick are converted to bit positions in a state variable. Each bit corresponds to one direction, using four bits in all. If a stick isn't present, it simply jumps to the section that looks at the buttons. After all that, the state of each button is determined and a bit in the state variable is set accordingly.

Note that near the top of the list of compares, I get the current state variable, shift it left twice and mask out all bits but bits 6 and 7. This is how I remember the previous state of the two buttons.

When all is done, I update the state variable.

Now why, you may ask, do I use a set of bits in a byte to remember direction? Well, the routine here is based on one I discovered when I was translating a program from the Commodore 64 to the Apple II. The C64 uses a digital joystick (one in which each direction is determined by a switch in the stick itself) and the switches in the stick are converted directly to bits within a status byte. It was more convenient to continue using the bits approach so I didn't have to modify a significant portion of the program being translated. I have since found it useful to continue to deal with the Apple joystick this way because I like it that way; it's neat and clean.

`DoJoystick`

This is the routine in which keyboards are read, buttons events are processed, and vectors are determined. Although it is quite feasible to use the data from UpdateJoystick directly (the handling of the status byte is a lot like handling the status byte from reading the mouse), I use DoJoystick to do the work for me. What this routine does is convert the bits in the status byte to vectors in the x and y directions. These vectors can then be added directly to a cursor position to change it in the direction the stick is pushed. Also, the state of the buttons is used to return event codes such as button up and button down, which are useful in mimicking mouse button events if you have a program that can use both input devices. And in addition to all of that, it calls DoKeystick to handle keyboard equivalent joystick moves.

DoJoystick should be called once per pass through your main loop and it should be called after UpdateJoystick. See the section on implementation for further details.

The vectors produced by DoJoystick are two variables, one for the x direction (left and right) and one for the y direction (up and down). The vectors are set to -1, 0 or +1 depending on the direction of the stick. By simply adding these vectors to the cursor position, you get a change in position (or not if the stick is centered in that direction). Two other variables are produced: one that tells the code of the button event that just happened and the other which is set to minus if a button is pressed else it is set to plus (this gives a quick test of a button down event). The button event codes are

0 = no button pressed
1 = button down event
2 = button up event
3 = button still being held down

Note that in this version of DoJoystick, I treat both buttons as one button, again to make it easier to mimic a mouse button.

`DoKeystick`

This routine is called by DoJoystick to see if any keys that mimic the joystick have been pressed. This version of DoKeystick polls the keyboard directly. It is entirely possible to replace the keyboard location with a variable you supply from your own program that holds the last keypress from your own keyboard polling routine. Note that you will have to change the wait for keypress loop for pausing (or cut it out altogether).

In any event, the table at the end of the routine holds the keypresses that are recognized as equivalents to the joystick. Currently, they are defined as WERSDZXC in a diamond shape. Note that the high bit is cleared on these. Also note the presence of two other keys, F and M which will mimic a button press and a combination button press and stick movement (as an example of how to do it) respectively.

Two other keypresses are currently supported in line in the routine. These are P for pause and control-J which will call InitJoystick. The pause simply waits for another key before exiting. Control-J is used when the joystick has become unplugged during the program's execution (in which case, UpdateJoystick will have set a flag saying the stick is no longer present) or when a joystick has been plugged in after the program is running (same deal with UpdateJoystick). By pressing Control-J, the initialization procedure is done which resets the internal flag determining the presence of the stick.

By adding keypresses to the table, you can mimic any joystick action or combination of actions. Note that key equivalents are always available even if the stick isn't present. Also note that keypresses take precedent over joystick actions.

And finally, in this version, the keyboard strobe is cleared if a recognized key is processed. Otherwise the keyboard strobe is not cleared leaving the keypress in the keyboard port so another routine can read it. This allows the presence of more than one key read routine.

`InitJoystick`

This is the very first routine your program needs to call before accessing the others. It will determine what type of Apple II the routine is being run in, initialize the internal variables used by the other routines, and will call UpdateJoystick to determine the presence or absence of a joystick. It will then return a value in the A register so the calling program can act on the absence (or presence) of the joystick. The value returned has the negative flag set or cleared (test only for that flag): set if the stick is missing, cleared if it is present. It is suggested that the calling program not take specific action if a stick is missing; let the joystick routines handle that problem automatically. This way a user can add a stick during the program's execution, press control-J and begin using the stick as though the program started out that way. This gives the best transparent operation of an input device. In any event, always calling DoJoystick will allow the keyboard equivalents to operate properly so you shouldn't have to filter calls to it if the stick isn't present. I supply the value to the calling routine so it can take possible action prompting the user that a stick isn't present and how to take alternative action.

`Apple_ID`

This routine simply identifies what type of Apple II computer the routines are being run in. It sets a variable to a number from 1-7 which identifies the machine:

1: Apple II	5: Apple III (emulation)
2: Apple II+	6: Apple IIc
3: Apple IIe	7: Apple IIgs
4: Apple IIe (enh)	255: unknown Apple

Further discriminations on individual machine types aren't necessary for the joystick routines.

How to use my routines

Okay, enough of the boring details. Now on to how to actually use these things!

Incorporating these routines into your own programs is really quite simple. These routines are best dealt with as a separate file that is linked into your own code. This avoids any problems

with possible duplicate labels and other conflicts. You may need to establish four variables in such a way that the joystick routines and your main program can access them equally. If not then you will have to indicate that these variables are external to your code. See the next section for more details. At the beginning of your program file or files, you add only three externals (if you are going to link the joystick routines into place), more if you are going to access the variables in the joystick file:

```
InitJoystick        ext
DoJoystick          ext
UpdateJoystick      ext
```

Then you call these as needed. See the section on how to call these routines for more details. The joystick file is completely self-contained as it stands now so all you have to add to your linker command file (if you need to) is:

```
asm  joystick8.s
```

and add to the linking section:

```
lnk  joystick8.l
```

to make the file part of your program.

Once the code is in place, you need to do something with the results that come out of the routines. That is covered in the section dealing with results.

What variables are needed by Custom Joystick

Custom Joystick uses five variables that a calling program will need to know about. These variables are already included in the joystick file and are marked as entry points. You can move these to your own equates file and put at the beginning of the joystick file a PUT EQUATES command so the joystick routines can gain access to those variables. The variables needed are:

joyvectx : a byte which holds the current vector for the x direction

joyvecty : a byte which holds the current vector for the y direction.

trigger : a byte whose high bit determines whether a button was pressed or not.

button_state : a byte which holds a code based on what type of button event last happened.

machine_id : a byte that indicates what type of Apple II the routines are being run in. This does not need to be communicated to the main program unless you wish to use it.

Calling procedures

Using these routines is very straightforward. The very first routine you need to call is InitJoystick. This sets up the joystick routines for action and determines the type of Apple being run on (a code for which is stored in machine_id). You need only do this once. InitJoystick will return a value in the A register which indicates whether or not a joystick is present. If the value is negative (i.e., the high bit is set) then there is no joystick present. Otherwise, the high bit will be clear indicating there is indeed a joystick to work with. I recommend not using this value in your own code unless you wish to explicitly handle the case where a joystick isn't present and needs to be or you wish to inform the user that it is possible to plug in a joystick while the program is running and then press control-J to activate the stick.

Okay, once the joystick has been initialized, you use the other two routines, UpdateJoystick and DoJoystick to actually use the joystick. UpdateJoystick will read the current state of the stick and put that information into a state variable internal to the routines. A call to DoJoystick will convert that state information into something your program can use. This is the routine that updates the four variables described earlier. Then your program can use the information in whatever way you have designed.

What to do with resulting information

DoJoystick returns four values stored in four variables. These variables are joyvectx, joyvecty, trigger, and button_state.

Joyvectx and joyvecty are vectors. These contain -1, 0, or +1 depending on the position of the joystick. By adding these to a cursor position, for example, you can cause that position to change based on the position of the stick. For example:

```
jsr  DoJoystick
lda  joyvectx
clc
adc  cursorx
```

```
    sta  cursorx
    lda  joyvecty
    clc
    adc  cursory
    sta  cursory
```

will update cursorx and cursory based on the last position of the stick. Note that you should add some code that checks for boundaries.

The use of trigger and button_state are equally as straightforward. For example:

```
    bit  trigger
    bmi  button pressed
    else button isn't pressed
```

and

```
    lda  button_state
    beq  no button pressed
    cmp  #1
    beq  button down event
    cmp  #2
    beq  button up event
    else button is still being held
down
```

Trigger is used for those times when you only need to know if the button is pressed or not, say, for a fire button in a game. Button_state is used for those times when more sophisticated situations call for more sophisticated button actions.

Differences between Custom Joystick and Joystick GS

The differences between the two principle versions of Custom Joystick are:

1) Joystick GS doesn't have Apple_ID in it since it assumes you will be running on a IIgs.

2) Uses machine instructions specific to the 65816 processor.

3) Requires the four variables, joyvectx, joyvecty, trigger, and button_state to be words instead of bytes (if you move them outside of the file).

4) Accesses the softswitches in the IIgs through Bank $E0.

Other than that, the logic of using the GS version is the same as for the 8 bit version.

```
 1          lst   off
 2          rel
 3          dsk   joystick8.1
 4
 5          xc
 6          xc
 7          mx    %11        ;all 8 bit code, thank you very much
 8
 9 * Requires the following labels external to this file (preferably direct
page):
10 *              (These should be byte values)
11 * trigger    = - if button is down
12 * button_state = state of button(s).
13 *                 0 = no button pressed        2 = button up
14 *                 1 = button down              3 = button still down
15 * joyvectx   = direction of x coordinate: -1, 0, +1.
16 * joyvecty   = direction of y coordinate: -1, 0, +1.
17 * machine_id = type of Apple being run in (set by apple_id in this file).
18 *
19 * These variables are defined here arbitrarily so the file can be assembled.
20 * See documentation on ways to deal with these variables.
21
22 trigger  ent
23          ds    1
24 button_state ent
25          ds    1
26 joyvectx ent
```

```
27          ds    1
28 joyvecty ent
29          ds    1
30 machine_id ent
31          ds    1
32
33
34 *-----------------------------------------------
35 * Joystick read routine (8 bit version) for the Apple II computer
36 * by Stephen P. Lepisto
37 * Date: 1/3/88
38 * Assembler: Merlin 16 v3.50+.
39 *
40 * Reads a standard analog joystick in a custom way.  Returns values that
41 * are 0-145 which is useful for vector-type motion.  Also reads the buttons
42 * and sets a global variable accordingly.  Combines both buttons into one.
43 *
44 * Note that these routines assume that there will be one call to dojoystick
45 * for every call to updatejoystick.  Updatejoystick adds to the state of the
46 * stick until dojoystick clears it so you can call updatejoystick more than
47 * once before you call dojoystick.
48 *
49 * Dokeystick: returns 0 if no joystick equivalent keys are pressed else
50 * returns a byte that looks like stickstate (see updatejoystick for
51 * specifics).
52 *
```

Custom Joystick will continue next month.

From Woz's new jokebook (*The Official Computer Freaks Joke Book, Bantam Books, $3.50*):

- CONSULTANT: Someone called in at the last moment to share the blame.

- A fellow had trouble with his head. A team of brain surgeons agreed to remove his brain, examine it, then put it back later. They performed the operation, but when they came to put his brain back, he wasn't there. The man had disappeared. A month later he returned to the happy doctors.

"Where have you been since we removed your brain?"

"I became a consultant at IBM."

- How many computer salesman does it take to screw in a light bulb?

"I'll have to get back to you."

- Keyboard Prayer:
Our program, who art in Memory, Hello by thy name. Thy Operating system come, thy commands be done, at the printer, as it is on the screen. Give us this day our daily Data, and forgive us our I/O errors as we forgive those whose logic circuits are faulty. Lead us not into frustration, and deliver us from power surges. For thine is the Algorithm, the Application, and the Solution, looping forever and ever. Return.

The Sourceror's Apprentice

American prices in US dollars effective January 1, 1989: 1 yr..$28, 2 yrs..$52, Canada and Mexico add $5, all others add $10

Back issues are available at $3.00 each.

Please direct all correspondence to:

Ariel Publishing, Inc.
P.O. Box 266
Unalakaleet, Alaska 99684 USA

THE APPRENTICE is a product of the United States of America.