The Sourceror's Apprentice

The Assembly Language Journal of Merlin Programmers Vol. 1 No. 4 April, 1989

The Month in Pictures

"What a dude I am! I finished two Apprentice's within just a few days of each other."

"Whaddya mean a bug?"

"Whaddya mean the printer gave me two page 13's?"

"Whaddya mean I left out TWO paragraphs out of an article?"

Was it the Ides of March?

I discovered one of the real pitfalls of editorship last month: complete and total public humiliation. It was not that bad, I suppose, but it sure felt like it for awhile. I pictured hundreds of people tossing their newsletters in the trash and writing me nasty notes. That would hurt even worse than canceled subscriptions. I've never been able to handle nasty notes very well. It's probably something left over from my grade school days.

Instead I got some very kind and generous treatment. Bob Sander-Cederlof called and offered encouragement and the comiseration that only a former newsletter editor could. Roger Wagner was more than patient. And Sandy Mossberg was a class act, too. Steve Stephenson (Genesis Software, author of the great Squirt series of program selectors) wrote me a very gentle note offering forth code to fix my boo boos. I thank you one and all.

As indicated in the graphic above, under deadline pressure I boogered up Generic Startup again. I think I'll let that dog sleep for awhile, except to say that if you send me a disk I'll put the repaired file on it and send it back to you at my expense. Quarterly disk subscribers already have the new version.

I am going to discuss the programming considerations in squashing the insectia in Generic Startup a little at a time and in another place: a new short monthly column I'm beginning called *The Gentleman's GS*. Aside from requesting attention to 8 bit programming techniques, the most common message I've received from subscribers is, "...programming the GS is driving me crazy. Please go slow and take it one step at a time."

Request granted. We'll start this month.

I also wanted to apologize to y'all and to Jerry Kindall for leaving two entire paragraphs out of his last installment of the *Applesoft Connection*. The missing paragraphs should come right before the sub-head "String Beans" and they read:

The next step is to call FREFAC. If we don't do that,

Applesoft will eventually run out of temporary string descriptors (which are used to keep track of strings generated while evaluating an expression) and give the unwary Applesoft programmer an ?OUT OF MEMORY error.

After calling these three routines in order, the length of the string is in the Accumulator, and a pointer to the string is in SPTR ($5E). The actual address of the string will vary depending on Applesoft memory conditions; SPTR gives us a way of finding it. From there, it can be accessed by your machine language program.

Like I said, it was a tough month.

One of my friends told me that he was downright irritated that I broke up Steven Lepisto's Vectored Joystick Programming source code. Steven's article and source code files were VERY long. I really had no choice, but be assured that I'll try to avoid it in the future. The rest of his 8 bit code is included in this issue. I hope you enjoy it.

The Applesoft Connection
Part 3: CALL & Ampersand

by Jerry Kindall
2612 Queensway Drive
Grove City, OH 43123
614/875-6805

GEnie: J.KINDALL
ALink: JKindall

In the first two parts of this series, I covered the nuts and bolts of passing parameters to and from BASIC. Now it's time to put all those little technical details together.

First, though, a couple of additions to article #2. Browsing through What's Where In The Apple shortly after submitting that article, I ran across a routine called FRESTR, a combination of CHKSTR and FREFAC. Since the string-reading technique I presented in the string discussion calls both CHKSTR and FREFAC in sequence, you can replace those two JSR's with one call to FRESTR at $E5FD, which will save you three whole bytes.

The other addition is a bug correction. The input-anything routine I presented works fine, if you tried it, but it doesn't check to make sure the variable it's been passed is a string; if you passed it a numeric variable, it wouldn't work, but you also wouldn't get an error. I left out a call to CHKSTR. To fix this, just insert a JSR CHKSTR immediately after line 29 of the routine. Note that I did include CHKSTR = $DD6C at the beginning of the program, so you can just stick that line in. (How I managed to include the equate for that routine while forgetting to call it is beyond me!)

The CALL Statement

Applesoft programmers use the CALL instruction for everything from clearing a screen line (CALL -868) to entering the System Monitor (CALL -151) to fixing the Applesoft ONERR bug (CALL -3288). CALL is so flexible because it lets the programmer specify the starting address of the machine-code routine to be executed instead of being hard-wired to an address like HOME or PRINT.

Once a machine-language routine has been activated via CALL, it can send and receive variable values using the routines we've been talking about in the last two parts of this series. Look at the input-anything routine in the second part of this series for an example of a CALLable routine.

The first thing a CALLed routine will usually do is call CHKCOM ($DEBE) to "eat" the comma between the CALL address and the parameters. (If the first parameter to be read is a numeric value from 0-255, you can also use COMBYT at $E74C.) Without the comma, Applesoft can (and by Murphy's law, probably will) get very confused as to what address you REALLY mean with that CALL. Compare CALL 768,0 to CALL 7680 and you'll see what I mean.

Other than the initial comma, there's no particular trick to writing CALLable routines. The Applesoft entry points discussed in the first two parts of this series will do all the rest of the work for you, as long as they're called properly.

Mr. Ampersand

BASIC's ampersand command (&) can also pass control to a machine language routine anywhere in the computer. This is done by storing a JMP to the desired routine in locations $3F5-$3F7. When Applesoft sees an ampersand, it immediately JMPs to $3F5, which then JMPs to the desired routine. Because $3F5 is only used to JMP to an ampersand routine, it's known as the "ampersand vector" or sometimes the "ampersand hook". (Another example of a vector is $3D0, the DOS/ProDOS warmstart vector. A vector's sole purpose is to JMP to another location, so that even if that other location is changed, programs which call the vector will still work.)

With the ampersand, you don't need to "eat" a comma at the beginning of your routine. To turn the input-anything routine into an ampersand routine, all you have to do is remove the JSR CHKCOM (line 28). (Move the label INPUTANY to the next line.) Then, to "activate" the routine, store a JMP $300 in the ampersand vector. (From the Monitor, just type 3F5:4C 00 03.)

When you enter & A$, Applesoft will jump to $3F5, which will jump to $300, the start of our routine. From there everything proceeds as usual.

Hook It Up

Most well-written ampersand routines contain code to hook themselves up. This way you don't have to hook them up by hand by going into the Monitor or with POKEs. This is done by adding a "front end" to the ampersand routine, which stores the appropriate JMP at $3F5-$3F7 and exits via an RTS. This means that ampersand routines must usually be BRUN to install them, whereas a CALLable routine can simply be BLOADed.

Let's add such a front end to our input-anything routine. Just after the ORG, add the following code:

```
28   HOOKUP     LDA  #$4C            ;JMP opcode
29              STA  $3F5
30              LDA  #INPUTANY       ;low byte of INPUTANY
31              STA  $3F6
32              LDA  #/INPUTANY      ;high byte of INPUTANY
33              STA  $3F7
34              RTS                  ;go back to BASIC
```

Now, when our input-anything routine is BRUN, this short front end gets executed, hooks up our ampersand routine to the ampersand vector, and exits to BASIC. The actual input-anything routine isn't executed until an ampersand is encountered in a program.

Since the front end isn't used after the routine has been installed, it can run in a non-permanent memory location. For example, we could have the above front end actually residing in the latter part of the keyboard buffer (normally a very unsafe place for a machine-language routine) as long as we made sure that our actual input-anything routine is safely in page 3.

Passing The Buck

But there's still a major problem with the ampersand version of our input-anything routine. (Don't you just love it?)

What happens if we want to install and use two different ampersand routines? (Let's assume for a moment that the two routines we want to use don't use the same area of memory. We'll deal with memory conflicts a little later.) Say we want to use both our input-anything routine and a string-swapping routine. We install the string-swapping routine first and it hooks itself up to the ampersand vector. Then we install our input-anything routine and it hooks itself up to the ampersand vector too. Now, when we enter an ampersand command, which routine will get control?

Well, since our input-anything routine was the last routine to install itself, it would get control. We have essentially lost track of any ampersand routine installed earlier. This gives Applesoft programmers huge Excedrin headaches as they try to figure out a way to allow all their ampersand routines to work together.

But don't be unduly consternated, there is a way to extricate ourselves from this quandary. When we install our ampersand routine, we can save the address of any previously installed ampersand routine. If the ampersand call isn't for our routine, we can pass it on.

This brings up another problem. How can we tell if an ampersand call is for us? The best way is to check for some character or series of characters immediately after the ampersand character. Since we're writing an input-anything routine, let's use the word INPUT. The advantage of using INPUT is that since it's an Applesoft command word, it's only one byte long, and so we can check for it with one instruction.

Therefore, if we find the word INPUT, we know the call's for us; otherwise, we pass it on to any previously installed ampersand routines. A typical call to our input-anything routine will look like & INPUT A$. Now that we've solved THAT problem, let's figure out how to avoid the other problem I glossed over just a minute ago.

Memory Conflicts

Page 3 of RAM is a popular area of memory for CALL and ampersand routines, and this is another thing that frequently causes Applesoft programmers to have headaches. Their favorite ampersand routines often compete for the same area of memory.

One solution to this dilemma is to ask BASIC.SYSTEM for some memory and move your routine there. Since we can't be sure when we're writing the routine exactly where BASIC.SYSTEM will put our memory, we have to make the ampersand routine relocatable. With our little input-anything routine, this is simple -- it's already relocatable, since we didn't reference any addresses within the program code itself.

If you can't easily make the routine relocatable, you will have to relocate it. See Karl Bunker's "Relocation Without Dislocation" in the February, 1989 issue of The Sourceror's Apprentice for much more information about relocating your programs.

There's only one drawback to this technique. Since BASIC.SYSTEM allocates memory 256 bytes at a time, our little input-anything routine will end up wasting about 200 bytes of space. Stick a half-dozen short routines up there and that wasted memory begins to add up. We won't worry too much about that, since there's little we can do about it, but it's something to be aware of.

Fruits Of Our Labors

The routine below is an ampersand-driven version of our input-anything routine. Take a look at it, then I'll detail exactly how it differs from our first version.

```
1    * The Applesoft Connection
2    * Yet Another Input-Anything - v2.0
3    *
```

```
 4   * Ampersand-driven, self-installing version
 5   * Jerry Kindall -- March, 1989
 6   *
 7
 8   * Page 0/2/3 Locations
 9
10   PTR      =     $1A         ;temporary pointer
11   PROMPT   =     $33         ;prompt printed by GETLN
12   VARPNT   =     $83         ;pointer to string variable
13   DSCTMP   =     $9D         ;holds new string descriptor
14   BUF      =     $200        ;keyboard buffer
15   AMPVEC   =     $3F5        ;ampersand vector
16
17   * Applesoft, Monitor, and BASIC.SYSTEM Routines
18
19   CHRGET   =     $B1
20   CHRGOT   =     $B7
21   GETBUF   =     $BEF5       ;allocate memory
22   OLDHI    =     $BEFB       ;original HIMEM MSB
23   GDBUFS   =     $D539       ;fix input buffer for BASIC
24   CHKSTR   =     $DD6C       ;check for string
25   CHKCOM   =     $DEBE       ;syntax check for comma
26   PTRGET   =     $DFE3       ;get pointer to variable
27   STRINI   =     $E3D5       ;init string space & pointer
28   MOVSTR   =     $E5E2       ;move string to string pool
29   GETLN    =     $FD6A       ;get an input line
30
31            ORG   $2000
32
33   * Installation Routine
34
35   INSTALL  LDA   #1          ;we need 1 page of RAM
36            JSR   GETBUF      ;allocate it
37            BCS   :EXIT       ;no RAM available
38            STA   OLDHI       ;protect our memory
39            STA   PTR+1       ;set up pointer for move
40            LDA   #0
41            STA   PTR
42
43            LDY   #END-INPUTANY ;move code up in memory
44   :LOOP    LDA   INPUTANY,Y
45            STA   (PTR),Y
46            DEY
47            BPL   :LOOP
48
49            LDY   #PASS-INPUTANY+4 ;save old amper vector
50            LDA   AMPVEC+1
51            STA   (PTR),Y
52            INY
53            LDA   AMPVEC+2
54            STA   (PTR),Y
55
56            LDA   #$4C        ;hook up our routine
57            STA   AMPVEC
58            LDA   PTR
59            STA   AMPVEC+1
60            LDA   PTR+1
61            STA   AMPVEC+2
62   :EXIT    RTS
63
```

```
64   * Input-Anything Routine
65
66   INPUTANY CMP   #$84        ;check for INPUT token
67            BNE   PASS
68            JSR   CHRGET      ;skip to next character
69
70            JSR   PTRGET      ;find variable
71            JSR   CHKSTR
72
73            LDA   #$80        ;control-@
74            STA   PROMPT      ;no prompt
75            JSR   GETLN       ;get input line
76            TXA               ;save string length
77            PHA
78
79            JSR   GDBUFS      ;make good Applesoft string
80
81            PLA               ;remember length
82            JSR   STRINI      ;make space & descriptor
83
84            LDX   #BUF        ;get address of input buffer
85            LDY   #/BUF
86            JSR   MOVSTR      ;move it to string pool
87
88            LDY   #2          ;move descriptor to variable
89   :LOOP    LDA   DSCTMP,Y
90            STA   (VARPNT),Y
91            DEY
92            BPL   :LOOP
93
94            RTS
95
96   PASS     JSR   CHRGOT      ;re-get character
97            JMP   PASS        ;this gets modified
98
99   END      =     *
```

The first real difference is at line 31. Instead of having an ORG of $300 as did our original routine, the new routine has an ORG of $2000. $2000 is a pretty standard load address for ampersand routines, probably because that's also the standard load address for SYS files. The memory at $2000 is no longer needed after the routine has actually been installed.

Lines 35-37 request one page (256 bytes) of RAM from BASIC.SYSTEM. This is the minimum amount of memory you can ask for. If the carry flag is set on return, no memory is available, so we exit without installing our routine. Otherwise, the accumulator holds the high byte of the address of our memory. In line 38, we store this in BASIC.SYSTEM's "original HIMEM MSB" location. This protects our routine in case some doofus comes along and does a FREBUFR call, which would otherwise free up the memory where our routine resides and allow it to be overwritten.

After doing this, we set up a pointer to our memory space and move the actual input-anything routine to our memory (lines 39-47). Then lines 49-54 copy the address of the old ampersand routine to line 97's JMP statement so that unrecognized ampersand routines can be passed to the next one in the chain. This is an example of self-modifying code, a practice which is generally frowned upon, but in this case it's the easiest way to do what we're trying to do. Since we don't know where the routine will end up, we have to figure out where that JMP statement is during runtime, which is done using a pointer and an index register. After doing all that, we hook up the ampersand vector to point to our newly installed routine (56-61), then we exit to BASIC.

The only thing different about the input-anything routine itself is in lines 66-68. When an ampersand routine is called, the accumulator contains the character that follows the ampersand. In this case, we're checking it to see if it's INPUT. If it's not, we exit through PASS (line 96), which calls CHRGOT (to reset the various status flags which CHRGET sets and our CMP instruction changed) and JMPs to the next ampersand handler on the chain. If it's our routine, we call CHRGET to advance to the next character in the statement. From there things proceed as usual.

More Still To Come!

In the next article in this series, I'll discuss the seriously underused USR function, which is powerful in a whole different way from the ampersand command. Until then, don't let the bit-bugs byte!

Random Bytes

Robert C. Moore
The Johns Hopkins University
Applied Physics Laboratory
Laurel, Maryland

Professor Moore works in the Space Department in his lab. His handiwork has flown on board the Apollo-Soyuz mission and will be aboard the GALILEO mssion to Jupiter. A multi-talented person, Bob is also an outstanding pianist.

Not only are Bob's random number generators statistically sound (as far as this non-mathematician can ascertain), but the two RANDOM routines are completely relocatable and easy to use. With just a little careful planning when you define your equates, MULTIRAND is also easily relocatable.

Frequently an assembly language programmer needs a random number. Random numbers are useful in simulations, where statistical "Monte Carlo" techniques are used. They also are useful in educational programs in which particular data (e.g., multiple choice answers) are selected at random from a large pool of possibilities. Many random number generators are available; however, most of them have serious problems.

The most notorious problem with random number generators is that many of them do not produce sufficiently random outputs. Some of them begin to repeat the sequence of random numbers after only a short while. These are said to have too short a period. Others have output numbers that are not well-distributed over the possible range of values. For these, certain values or patterns of values are much more likely to occur than other values or patterns.

Fortunately for Apple II assembly language programmers, several good random number generators exist. Among these are two highly recommended routines: one published by Don Lancaster [Ref. 1] and another published by David G. Sparks. [Ref. 2] Both of these programs are excellent. Lancaster's program requires 35 bytes and generates a random byte in 374 microseconds; Sparks's program, which is compatible with Applesoft, requires 256 bytes and executes in approximately 27.9 milliseconds. Sparks's program generates a random real variable value for Applesoft BASIC.

I have developed a pseudo-random number (PRN) generator subroutine that is fairly well-behaved (its period is greater than thirty million), very short (as few as thirty-one bytes), and very fast (67 microseconds on a standard Apple IIe or //c). It uses the technique of generating a pseudo-random sequence with a feedback shift register.

Figure 1 illustrates the shift register concept. A 25-bit shift register is shown; it can shift its contents to the left. The input data for the rightmost bit of the shift register are supplied by exclusive-ORing data bits from the register itself. By selecting data bits 7 and 25 (see Figure 1) for feedback to the input of bit 1 (i.e., on the next shift BIT1 will receive BIT7 EOR BIT25), a maximum length sequence of register data will be generated, provided the initial contents of the shift register is not zero. [Ref. 3] A maximum length sequence

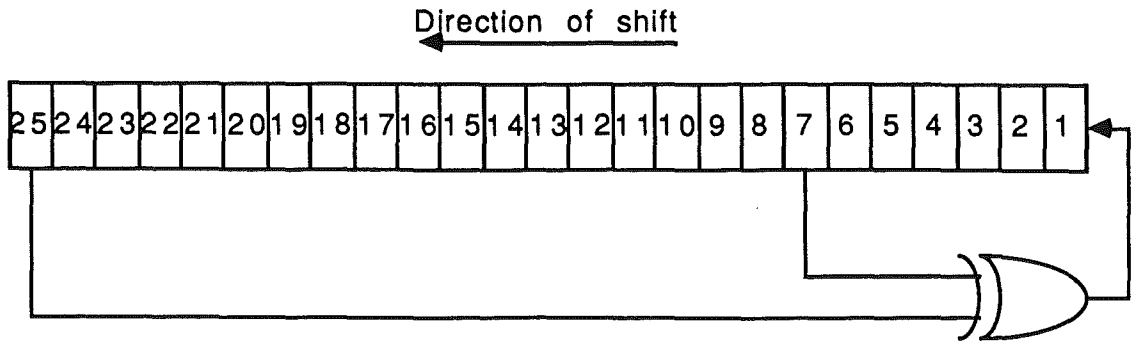


FIGURE 1: A 25-BIT PSEUDO-RANDOM NUMBER (PRN) SHIFT REGISTER

has a period of 2^n-1, where n is the number of bits in the shift register. For n=25, 2^n-1 is 33,554,431. This period is sufficiently large for most random number applications.

The period of this particular PRN shift register, 33,554,431, has the following integer prime factors: 1801, 601, and 31. If we shift the PRN register once each time we wish to generate a new bit, the period of the resulting bit sequence will be 33,554,431.

However, we wish to generate random bytes, not just random bits. Let's say we take our random byte from the rightmost eight bits of the PRN register. By shifting only one bit at a time, seven of those eight bits will be identical (but shifted one bit to the left) to seven of the bits in the most recent random byte. This correlation between successive bytes is not good. If, however, we shift by more than one bit at a time, the correlation between successive random bytes will be reduced significantly. If the number of shifts is coprime with the PRN register period (i.e., it does not equal any of the integer prime factors of the PRN register period), the overall period will remain the same; that is, all possible 25-bit patterns (except the all-zero pattern) will occur before the first "seed" pattern repeats.

For my design I chose to shift the register seven times each time I request a new random byte. The reasons for this choice will become apparent when you study the subroutine. This means that only one bit of the new number is retained from the previous one. Following seven shifts, the PRN register of Figure 1 will look like the one shown in Figure 2, where the original bit numbers have been preserved.

This will result in an acceptable (to me, at least) sequence of random bytes. But how do we implement such a structure in assembly language? Here's my method. First, I set up four one-byte integer variables in which to store my PRN data. (I'll be using only 25 of the 32 bits in these four bytes.) Variable R1 will hold bits 1 through 8 of the PRN shift register, with bit 1 in its least-significant bit. Similarly, variable R2 will hold PRN bits 9 through 16, variable R3 will hold PRN bits 17 through 24, and the LSB of variable R4 will hold PRN bit 25. This is illustrated in Figure 3.

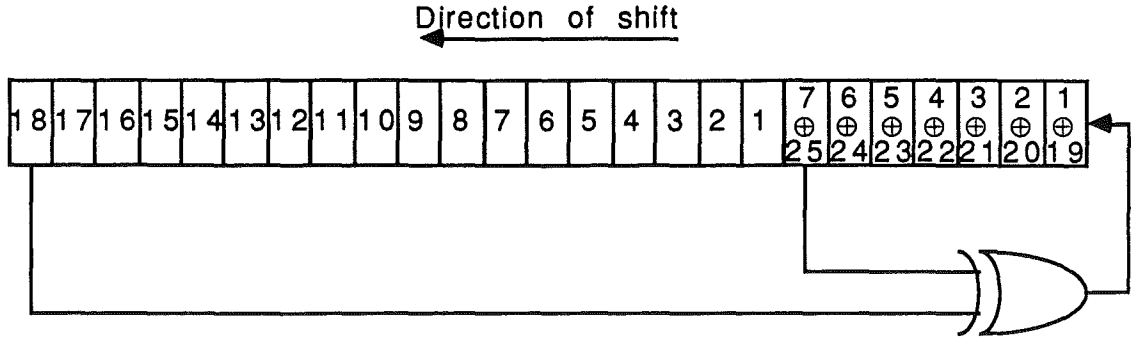


FIGURE 2: PRN SHIFT REGISTER FOLLOWING SEVEN SHIFTS

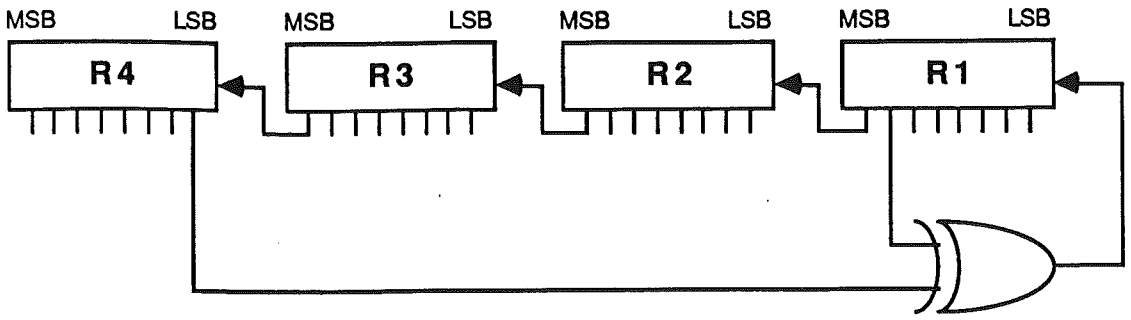


FIGURE 3: PRN REGISTER IMPLEMENTED IN FOUR 1-BYTE VARIABLES
PERIOD = 33,554,431

Now we can implement the PRN algorithm with the following sequence of assembly language instructions:

```
RANDOM ROR R4    ;Bit 25 to carry
       LDA R3    ;Shift left 8 bits
       STA R4
       LDA R2
       STA R3
       LDA R1
       STA R2
       LDA R4    ;Get original bits
17-24
       ROR       ;Now bits 18-25 in AC
       ROL R1    ;R1 holds bits 1-7
       EOR R1    ;Seven bits at once
       ROR R4    ;Shift right by one
bit
       ROR R3
       ROR R2
       ROR A
       STA R1
       RTS       ;Return to caller
```

You can quickly verify that this simple routine will transform the condition illustrated by Figure 1 into the condition which is illustrated by Figure 2. Notice that the routine has no loops or branches; it just falls straight through to the RTS.

If variables R1 through R4 are located in page zero, this subroutine executes in 66.5 microseconds. With the variables in regular memory (perhaps stored at the end of the subroutine), execution time is 80.2 microseconds. These speeds are quite good. On return to the calling program, the random byte is found in variable R1 and the accumulator. The contents of the X and Y registers are not modified. The PRN register, variables R1 through R4, must be "seeded" once with any nonzero value prior to the first call to RANDOM.

If your application requires a longer period, you may wish to use the design that is shown in Figure 4. [Ref. 4] This PRN shift register has a period of 2,147,483,647. The following subroutine shifts the PRN register of Figure 4 eight times, then returns the new random byte in R1 and the accumulator.

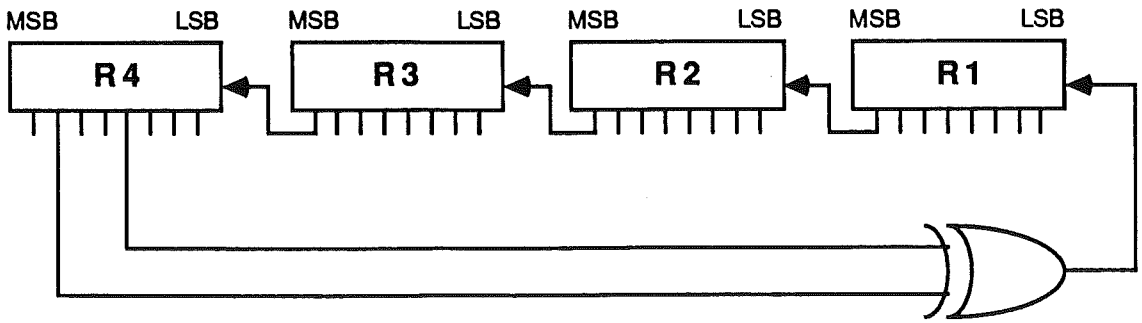


FIGURE 4: PRN REGISTER WITH PERIOD = 2,147,483,647
(From Reference [1].)

```
RANDOM LDA R4       ;Save original R4
       PHA          ; on the stack.
       LDA R3       ;Shift R3 left 8
bits
       STA R4
       PLA          ;Retrieve original
R4
       ASL R3       ;Shift left one bit
       ROL A
       PHA          ;Save bits 31-24
       ASL R3       ;Shift left three
bits
       ROL A
       ASL R3
       ROL A
       ASL R3
       ROL A
       PHA          ;Save bits 28-21
       LDA R2       ;Shift left 8 bits
       STA R3
       LDA R1
       STA R2
       PLA          ;Retrieve bits 28-21
       STA R1
       PLA          ;Retrieve bits
31-24
       EOR R1       ;Eight bits at once
       STA R1
       RTS          ;Return to caller
```

This program does not use the X and Y registers. It requires 39 bytes of program storage and executes in less than 100 microseconds. It implements the same PRN algorithm that Don Lancaster used, but executes in much less time than his implementation. (This is because Lancaster's implementation is more general-purpose than mine.)

The random byte that is returned by either of these RANDOM subroutines will be in the range from 0 through 255. Sometimes it is convenient to have random integers that are restricted to a narrower range. For example, to get a random integer in the range from 0 through 7, you could obtain a random byte and then mask off all but the least-significant three bits using an AND #$07 instruction.

But what if your narrower range does not extend from zero through 2^n-1? In that case a simple logic mask operation will not suffice. The following subroutine, MULTRAND, will work with either of the RANDOM subroutines described above. It will return a random integer in the range 0 through LIMIT-1, where LIMIT is the 8-bit integer parameter that is passed to the routine in the accumulator.

```
MULTRAND STA LIMIT      ;Use limit as
multiplier
         JSR RANDOM     ;Multiplicand
in R1
         LDA #8         ;Initialize
loop counter
         STA COUNTER
         LDA #0         ;Initialize
the product
SHIFMULT LSR LIMIT      ;Multiplier
bit to carry
         BCC SHIFPROD   ;If zero,
just shift
         CLC            ;Else add
multiplicand
         ADC R1         ; to the
product.
SHIFPROD ROR A          ;Shift the
product right
         DEC COUNTER    ;Decrement
loop counter
         BNE SHIFMULT   ;Go back if
not done
         RTS            ;Return to
caller
```

When the MULTRAND subroutine returns to the calling routine the COUNTER holds zero. The X and Y registers are unused and therefore their contents remain intact. The random byte, in the range from 0 through LIMIT-1 (LIMIT was the integer that was passed in the accumulator) is returned in the accumulator. The MULTRAND subroutine requires two local 8-bit integer variables: LIMIT and COUNTER, both of which are cleared to zero before MULTRAND returns control to its caller.

MULTRAND works by forming the integer product of LIMIT and a random 8-bit integer, R1, which is obtained by calling RANDOM. The most-significant half of the product is passed back to the calling routine; the least-significant half is discarded.

I have found these random byte generation subroutines to be very useful in writing assembly language programs for computer-aided education. Perhaps you, too, will find many uses for them.

REFERENCES

1 Don Lancaster, Assembly Cookbook for the Apple II/IIe (Indianapolis, Indiana: Howard W. Sams & Co., Inc., 1984) pp. 345-361.

2 David G. Sparks, "The Last Word on Random Numbers in Applesoft...," Call -A.P.P.L.E, vol. 12, no. 4, April 1989, pp. 44-47.

3 Neal Glover, Practical Error Correction Design for Engineers, (Broomfield, Colorado: Data Systems Technology Corporation, 1982), pp. 41-44.

4 Don Lancaster, op. cit.

The Gentleman's GS: A Polite Introduction to the 16 Bit II

I used to work at a wilderness camp modeled after the famous Outward Bound program. One of my favorite activities was when we took campers rappelling, which is essentially walking down a sheer cliff while attached to a climbing rope. The hardest part for all of our students was the first step backwards over the edge of the cliff. It was easy to understand why - you have to hang your buns out over a 100 foot drop, not something any of us are very used to.

Programming the GS is a little like rappelling. Few people are used to it. Like rappelling, though, I think that most people find it quite enjoyable once they get into it. You can go a tremendous distance with a little effort during a rappel. Likewise on the GS.

And like I used to do with our budding rappellers, I want to change your mindset: rappelling and the GS are both incredibly fun. The GS is not nearly as dangerous, either.

There are plenty of guides, too. As I've mentioned before, Gary Little's *Exploring the Apple IIGS* (Addison-Wesley) is a terrific introductory guide to the GS. So is Ron Lichty and David Eyes' *Programming the Apple IIGS in Assembly Language*. Both are first rate.

Still, a large number of **Apprentice** subscribers have requested a column such as this one to help clarify some of the less obvious issues. I know that Merlin programmers get frustrated always having to convert source code from APW to Merlin format. We'll help you through that process here.

And still other folks I know are struggling with all of the new terminology being bandied about. Think of it as meeting a few new (and powerful) friends. Let's proceed with the introductions...

A lot of people start getting sweaty at the mention of "event driven" programming. Relax. In some sense, all software with *any* user interface is event driven. Just keep in mind that the desktop metaphor means that the user should be able to do just about anything they wish anytime they want to. If you're sitting at your own desk, for example, you can move papers around, open a drawer, and go back to your paperwork with ease. So too, in the desktop interface scheme. From a programmer's standpoint, all it means is that we have to check the mouse and keyboard every so often to see what the human in control (henceforth known as HIC) wants to do. This occassional checking is often done in something called an "event loop" in GS lingo.

The only exceptions are activities that are impossible, illogical, or inconvenient for the user if interrupted (printing, for example). Such actions are called "modal" and should be avoided unless absolutely necessary. But they are necessary at times, so don't break out in hives worrying about them.

It should be encouraging, too, that Apple developed a toolbox routine called TaskMaster that checks and handles zillions of events for us automatically. One call does it all. TaskMaster doesn't even exist on the Mac (yet), so we have something they don't. Neener neener neener.

Ah yes, the toolbox. The notion of a toolbox shouldn't be intimidating either. Every time you've done a JSR COUT on an 8 bit Apple you've taken advantage of a routine that exists in your computer's ROM. In the same vein, the famous toolboxes are ROM routines. There's just more of 'em and they're quite a bit more sophisticated. They do more.

The increased sophistication and capabilities come at a price. One of the new costs is that the routines need data. One of the primary tasks for GS programmers is developing the sets of data required by particular tool calls. Then we must tell the tools where the data is. And since some tools send us messages back, we also have to create a place for messages. By convention, the toolbox sends and receives messages on the stack. Creating space for the toolbox messages involves pushing dummy arguments on the stack.

It is really pretty amazing to me the amount of work that one tool call can do. Remember my discussion of `AlertWindow` a couple of months ago? One call and your GS will display an alert with an icon and buttons. It will track the mouse and handle button selection, returning control to you with the number of the button selected on top of the stack. That seems more like a high level language to me than assembly code!

The toolboxes have a few other requirements, too. For example, they ALL need to be explicity started up at the beginning of a program. Unfortunately, you cannot run around and start up whatever you want whenever you want. Some of the toolbox routines are interdependent. These interdependencies were really the main motivation behind the now-mangled Generic Startup code we published. This was enough of an issue that Apple Developer Technical Support finally came out with the following chart in a tech note:

(You may start those tools which are indented at that time or any time thereafter. The numbers in paretheses are the tool numbers.)

```
Tool Locator
       ADB Tools              (#1)
       Integer Math Tools     (#9)
       Text Tools            (#11)
Memory Manager                (#2)
       SANE                  (#10)
       ACE                   (#29)
Miscellaneous Tools           (#3)
       Scheduler              (#7)
       System Loader         (#17)
QuickDraw II                  (#4)
       QuickDraw II Aux       (#18)
```

```
Event Manager                 (#6)
Window Manager               (#14)
Control Manager              (#16)
Menu Manager                 (#15)
LineEdit                     (#20)
Dialog Manager               (#21)
       either
       Sound Tools then       (#8)
       Note Synthesizer      (#25)
       or
       Note Sequencer then   (#26)
       MIDI Tools            (#32)
Standard File Operations     (#23)
Scrap Manager                (#22)
Desk Manager                  (#5)
List Manager                 (#28)
Font Manager                 (#27)
Print Manager                (#19)
```

The tech note adds: "Although you may start the sound-related tools any time after the Miscellaneous Tools, we recommend you start them after most of the Desktop-related tools."

The list above does not comprise ALL of the toolsets available, but it does outline the proper order for the most common.

Next Month: Generic Startup revisited.

Vectored Joystick Programming: 8 Bit Source (continued)

Refer to article in Vol. 1 No. 3, March, 1989

```
53 *-------------------------------------------------
54 *
55 * To use these routines:
56 * 1) call initjoystick to intialize the routines and determine if there is a
57 *    joystick present.  Only has to be done once.
58 * 2) at top of main loop, call updatejoystick to get current state of stick.
59 * 3) sometime after calling updatejoystick, call dojoystick to process state
60 *   of stick and return vector and trigger values.
61 *
62 * If stick isn't present, updatejoystick and dojoystick only process button
63 * presses (a la the apple keys).  If you wish, you can allow for installing a
64 * joystick on the fly by having the user press a key then based on that key,
65 * call initjoystick again.  If the stick is ever unplugged on the fly,
66 * updatejoystick and dojoystick will fall back to reading only buttons,
67 * leaving the stick itself in a centered state.
68 *-------------------------------------------------
69
70 * Hardware locations.
71
```

```
keypress equ   $c000      ;- if valid key press present
keystrobe equ  $c010      ;access to clear keypress
gs_speed equ   $C036      ;speed register of IIgs
resetstick equ $c070      ;reset paddle timers
rdstickx equ   $c064      ;timer for paddle 0 (+ when done)
rdsticky equ   $c065      ;timer for paddle 1 (+ when done)
button0  equ   $c061      ; - if button 0 pushed
button1  equ   $c062      ; - 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

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

* Initializes some variables and determines if stick is really plugged in.
* Returns A.reg=$FF00 if stick not available else A.reg = $0000.

initjoystick ent
          jsr   apple_id
          lda   #-1
          sta   stick_temp
          lda   #0
          sta   stickstate
          sta   trigger
          sta   stick live
          jsr   updatejoystick
          lda   stick_live
          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).

dokeystick
          lda   keypress
          bpl   :x
          cmp   #"a"
          bcc   :0
          cmp   #"z"+1
          bcs   :0
          and   #$df
```

```
:0
         and   #$7f
         cmp   #'P'
         bne   :0a
         sta   keystrobe
:waitkey
         lda   keypress
         bpl   :waitkey
         sta   keystrobe
         bmi   :x
:0a
         cmp   #$0a       ;ctrl-J
         bne   :1
         jsr   initjoystick
         jmp   :x
:1
         sta   dokey_char
         ldy   #-1
:2
         iny
         lda   key_table,y
         beq   :x
         cmp   dokey_char
         bne   :2
         lda   joyxlate_tbl,y
         sta   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


* 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 +.
```

```
193 *  stickstate : before next updatejoystick, current state of stick.  Bit 4
194 *               reflects current position of button, set if button down.
195 *
196
197 dojoystick ent
198           jsr   dokeystick ;read and interpret keys as joystick
199           bne   :1         ;branch if key equivalent pressed
200           bpl   :a         ;button equivalent key not pressed
201           lda   #0         ;else clear motion vectors
202           sta   joyvectx
203           sta   joyvecty
204           bit   stick_live
205           jmp   :6
206 :a
207           lda   stickstate
208 :1
209           sta   stickstate
210           lda   stickstate
211
212           cmp   stick_temp ;has the state changed?
213           beq   :6         ;branch if not
214           sta   stick_temp
215           ldx   #0         ;yes - which way?
216           ldy   #0
217           ror
218           bcc   :2
219           dey              ;up
220 :2
221           ror
222           bcc   :3
223           iny              ;down
224 :3
225           ror
226           bcc   :4
227           dex              ;left
228 :4
229           ror
230           bcc   :5
231           inx              ;right
232 :5
233           stx   joyvectx
234           sty   joyvecty   ;update state variables
235 :6
236           lda   stickstate
237           asl
238           asl
239           eor   stickstate
240           and   #%11000000
241           beq   :nochange
242           ldy   #2
243           lda   stickstate
244           and   #%00110000
245           beq   :skipchange
246           ldy   #1
247           bne   :skipchange
248 :nochange
249           ldy   #0
250           lda   stickstate
251           and   #%00110000
252           beq   :skipchange
```

```
253           ldy   #3
254 :skipchange
255           sty   button_state
256           tya
257           lsr
258           bcc   :8          ;button not down
259 :7
260           lda   #-1         ;else indicate button was pressed
261           bmi   :9
262 :8
263           lda   #0
264 :9
265           sta   trigger
266           lda   stickstate
267           and   #%00110000
268           sta   stickstate
269           rts
270
271
272 * Get values from joystick and convert to bit positions
273 * in 'stickstate'.
274 *
275 * The "dead space" around center is about 65%.
276 *
277 * Output:
278 * 'stickstate'
279 * bit 0 : = 1 if stick is up
280 * bit 1 : = 1 if stick is down
281 * bit 2 : = 1 if stick is left
282 * bit 3 : = 1 if stick is right
283 * bit 4 : = 1 if button pushed
284 * bit 4 : = 1 if button 0 pushed
285 * bit 5 : = 1 if button 1 pushed
286 * bit 6 : previous state of button 0
287 * bit 7 : previous state of button 1
288
289 updatejoystick ent
290           bit   stick_live
291           bmi   :5
292           jsr   readstick
293           cpx   #255
294           bne   :1
295           lda   #-1
296           bmi   :1a
297 :1
298           lda   #0
299 :1a
300           sta   stick_live
301           lda   stickstate
302           and   #%00110000 ;isolate button bits
303           asl              ;shift them to bits 6 and 7
304           asl
305           bit   stick_live
306           bmi   :5
307           cpy   #16         ;up
308           bcs   :2
309           ora   #%00000001
310 :2
311           cpy   #100        ;down
312           bcc   :3
313           ora   #%00000010
```

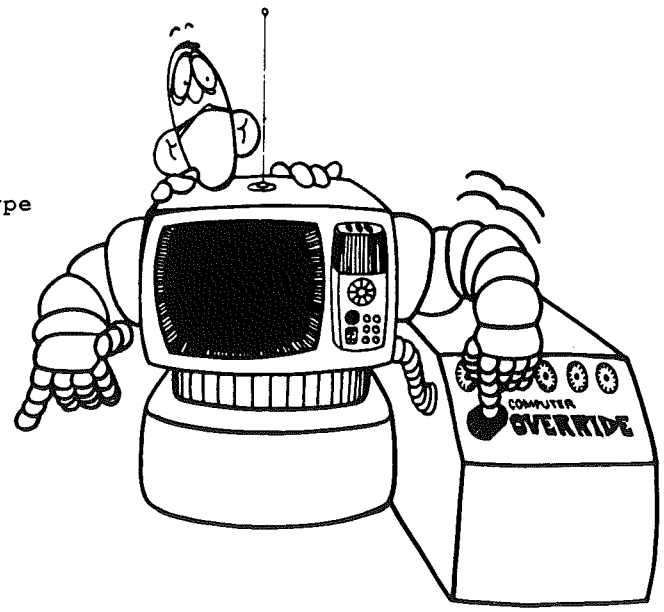
```
314 :3
315          cpx   #16        ;left
316          bcs   :4
317          ora   #%00000100
318 :4
319          cpx   #100       ;right
320          bcc   :5
321          ora   #%00001000
322 :5
323          tax
324          lda   button0
325          bpl   :6
326          txa
327          ora   #%00010000
328          tax
329 :6
330          lda   button1
331          bpl   :7
332          txa
333          ora   #%00100000
334          tax
335 :7
336          stx   stickstate
337          rts
338
339
340 * Read apple joystick, returning values for left/right, up/down directions.
341 *
342 * Output:
343 * x.reg = value for horizontal movement (0-145)
344 *       = 255 if no stick is attached.
345 * y.reg = value for vertical movement (0-145)
346 *
347 * Timing: minimum (both x,y read 0) = 90 (116) cycles
348 *   maximum (both read 145) = 3992 (4018) cycles
349 *   If no stick plugged in = approx. 6989 (7015) cycles
350 * Times in parentheses are for IIgs (extra time required for saving speed).
351
352 readstick
353          php
354          sei
355          jsr   slow_down
356          lda   resetstick ;reset timers on all paddles
357          ldx   #0
358          ldy   #0
359 :1
360          nop              ;delay tactics to compensate for
361          nop              ;the inx/bne :2
362          nop
363 :2
364          lda   rdsticky
365          bpl   :4         ;branch if done reading
366          iny
367          beq   :5         ;escape hatch if stick not plugged in
368          lda   rdstickx
369          bpl   :1         ;branch if done reading
370 :3
371          inx
372          bne   :2         ;always branches (it had better!)
```

```
373 :4
374          nop
375          nop
376          nop              ;compensation for not doing the iny/beq :5
377          lda   rdstickx
378          bmi   :3         ;branch if still reading
379 :5
380          jsr   speed_up
381          plp
382          rts
383
384
385 slow_down
386          lda   machine_id
387          cmp   #7
388          bne   :x
389          lda   gs_speed
390          sta   oldspeed
391          and   #$7f
392          sta   gs_speed
393 :x
394          rts
395
396 speed_up
397          lda   machine_id
398          cmp   #7
399          bne   :x
400          lda   oldspeed
401          and   #$80
402          ora   gs_speed
403          sta   gs_speed
404 :x
405          rts
406
407 oldspeed ds    1
408
409
410 * Apple ID routine
411 * Date: 3/28/88
412 * by Stephen P. Lepisto
413 *
414 * This routine will identify which Apple it is currently
415 * residing in and set a global machine ID byte
416 * appropriately.  It will also determine various things about
417 * what the machine is capable of.
418 *
419 * machine_id (1-7):
420 *   1 = II        2 = II+        3 = ///        4 = IIe
421 *   5 = eIIe      6 = IIc        7 = IIgs
422 * 255 = unknown machine
423
424 idbyte_1 equ   $fbb3
425 idbyte_2 equ   $fb1e
426 idbyte_3 equ   $fbc0
427 idrtn    equ   $fe1f      ;call for IIgs and eIIe (c=1 for eIIe)
428 ii_1     =     $38        ;always Apple II
429 iiplus_1 =     $ea        ; always Apple II+
430 iiplus_2 =     $ad        ;/
431 iii_1    =     $ea        ; always Apple /// (emulation mode)
432 iii_2    =     $8a        ;/
433 notiiplus_1 =  $06        ;must be IIe, eIIe, IIc, or IIgs
```

```
iie_3     =     $ea         ;always IIe
eiie_3    =     $e0         ;must be eIIe or IIgs
iic_3     =     $00         ;always any IIc
iic_4     =     $ff         ;original IIc
iic35_4   =     $00         ;IIc with 3.5 ROM
iicex_4   =     $03         ;IIc with memory expansion
iicrev_4  =     $04         ;IIc with revised mem expansion
iigs_3    =     $e0         ;must be IIgs or eIIe

apple_id ent
         lda   idbyte_1
         cmp   #notiiplus_1
         beq   :1          ;must not be a II, II+, or III
         ldx   #1          ;II
         cmp   #ii_1
         beq   set_id      ;is a II
         inx               ;II+
         cmp   #iiplus_1
         bne   unknownapple
         lda   idbyte_2
         cmp   #iiplus_2
         beq   set_id      ;is a II+
         inx
         cmp   #iii_2
         beq   set_id      ;is a III
         bne   unknownapple
:1
         ldx   #6          ;IIc
         lda   idbyte_3
         beq   set_id      ;is some form of IIc
         ldx   #4          ;IIe
         cmp   #iie_3
         inx               ;eIIe
         beq   set_id      ;is IIe
         cmp   #eiie_3
         bne   unknownapple
         sec
         jsr   idrtn
         bcs   set_id      ;is eIIe
         ldx   #7          ;IIgs
set_id
         stx   machine_id
         rts

unknownapple
         ldx   #255
         bne   set_id      ;unknown apple type
```



The Sourceror's Apprentice

Until May 28, 1989:

P.O. Box 266
Unalakleet, AK 99684

After May 28th:

P.O. Box 398
Pateros, WA 98846

Our phone after May 28th will be 509/923-2025.

American prices in US dollars:

1 year..$28 2 years..$52

Canadian and Mexican subscrbers add $5 per year. All other non-North American subscribers add $15 per year (covers first class postage).

Editor and Publisher..... Ross W. Lambert
Technical Editor..... Eric Mueller
Technical Editor..... Jay Jennings
Subscription Services..... Cindy Eckels
Stamp Licking..... Rebecca Lambert

Ariel Publishing
P.O. Box 266
Unalakleet, AK 99684

BULK RATE
U.S. POSTAGE PAID
Ariel Publishing, Inc.