The Sourceror's Apprentice

The Assembly Language Journal of Merlin Programmers Vol. 1 No. 8 August, 1989

Jerry, Jeff, and Junk Mail

First, the good news: Jerry is back! Jerry Kindall, author of our popular Applesoft Connection series, managed to crank out another magnificent treatise for us this month. Additionally, Jeff Smith, a member of Roger Wagner's technical support staff, granted us a very useful insight into the inner workings of the ERR.USR function in Merlin 16+. His article can really help you get more out of Merlin and your GS.

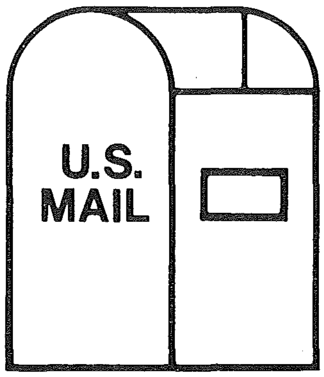
Now the bad news: I boogered up SAPP.DISPLAYER again on the quarterly disks (sigh). What was intended to be a quick little "convenience" program has transformed itself into a raging monster. SAPP DISPLAYER's problems notwithstanding, the quarterly disk contains TXT files readable by any ol' word processor and directly incorporatable into Merlin itself. Uh, almost - several of my formatted files (for typesetting) snuck in there in the guise of source code files. This is not fatal, you'll just have to remove the tabs. My sincere apologies. As recompense, I have decided to extend everyone's disk subscriptions by one quarter. You got the last one for free.

Incidentally, the quarterly disk is **not** a bootable disk - it is data only.

I Knew It Was Too Good To Last...

We here at the Ariel Cabin were purchasing Chris Haun's DesignMaster directly from him. We were not publishing the software ourselves, though I have some regrets that I did not try harder to make that happen (hey, I'm far too busy already!). To make a long story short, I at least have an eye for a fine piece of work: Chris is currently negotiating with a major publisher, hence our plans to sell the software are on hold. I am sorry for y'all, our customers, but I'm happy for Chris and the rest of Appledom.

By the time you read this you should have received the first ever Ariel Publishing Junk Mailer. It includes the prices on the Applied Ingenuity hard drives I mentioned last month (but did not include with the newsletter).



I will try to keep ads out of the *The Apprentice* proper since our space is always at a premium. Hence the junk mailer. Let me assure you that there 'tain't no junk involved, though. Just check out the products and the prices we were able to extort from a few select companies!

My intro has no flow to it this month, anyway, so I might as well plop in a quick tip right here. Eight bit folks: I shall make my point in the context of GS programming, but the principle applies anywhere.

This is no major revelation, I'm sure, but I have trouble figuring out why folks define data like this:

```
PenPos adrl 0
```

and then have to add offsets to access fields within the data, like so:

```
         lda PenPos+2   ; gets horizontal pos
```

Presumeably, the reason is that QuickDraw needs the pointer to the beginning of the entire data record, but you can generate more descriptive labels and accomodate QuickDraw like so:

```
PenPos
PenVert ds 2
PenHorz ds 2
```

Since Merlin (and APW) let you have multiple labels for the same spot, QuickDraw is appeased *and* you can still address your fields by name. Of course, you have a little extra typing when you define your data record. But I prefer using descriptive, easy to remember labels throughout my code. It is worth the extra overhead to me.

System 5.0 Info???

I have in hand the early release documentation for System 5.0. Data regarding the new "StartTools" , "ShutDownTools", and "TextEdit" calls is conspicuously absent. I'll chase it down post haste, but try not to be over anxious (I've had several phone calls), the reason being that preliminary documentation is notoriously inaccurate and subject to change.

Some Credit Where Due

I berated Apple, Inc. 's marketing department last month - and I standby my comments. But I want to give some credit to Cambridge Marketing, the producers of AppleFest (who are undoubtedly influenced by Apple, Inc.). Cambridge caught some flak from Apple II developers by encouraging Mac products at AppleFest and by seeming to ignore smaller companies (like us).

Well, Cambridge heard your outcries and responded in a most positive manner. Not only did they seek out Ariel Publishing (they called *me*), but they sent a mailer to all the small developers they could find (but don't ask me how they figured out who was small and who was "bigtime"). At any rate, they have been extremely flexible , available, and accomodating. Their attitude has been, "We want you there. What is it going to take to make that happen?"

I was going to pass on AppleFest SF this year, but thanks to Cambridge, we'll be there. Come see us in booth #844.

To Err is Human, To ERR.USR is Divine

Jeff Smith
1525 Graves Ave. #141
El Cajon, CA 92021

Editor: The new Merlin 16+ is a really ***big*** *package. I am still digging through all of the new features and support files. I was overwhelmed enough, initially, that I didn't give Jeff's article the attention it deserved. I regret that, because I think all of us (well, especially me, anyway) would be working a little smarter right now if I had. This ERR.USR thingamajig is slickern anything!*

Working with the Toolbox can be confusing sometimes, especially when an error happens. Both HodgePodge and Generic StartUp capture errors by tossing the error number and a meaningless value for X to the SysFailMgr. This leaves you staring at that obnoxious zipping apple message, and you can't do diddly except reboot. By setting X to different values everywhere, you can sort of figure out where the code went wrong. (As long as you don't forget where you were in the count...)

Wouldn't it be nice (as much as I hate to admit it) to go back to the bad old days of AppleSoft, where programs not only gave an error, but also gave the exact line number where the error occurred? Might there be be some way to get X set to the line number of your Source Code?

And, while we're making wishes, let's add a couple more. How about the ability to print the actual file name of the source code, for those of us that are working with Link files? And maybe we could find some way of displaying the secondary line number for errors in Put files. And, most off all, put the bloody thing in a dialog box so that pressing any key sends you back to Merlin or your program launcher instead of locked onto never-never land.

Further enhancements immediately come to mind. Perhaps a way for it to display its message even when the dialog manager isn't started up. And there ought to be a simple way, using conditional assembly, to have it remove every trace of this error stuff when the program is all polished off and perfect.

Putting it all that way, I can't believe anyone out there programs without ERR.USR!

HOW TO USE IT:

Note: Unfortunately, Merlin.16+ uses a very different format for its USR files than it used to, now that it's truly a 16 bit program. Thus, this routine ONLY works with Merlin.16+.

1. Since Merlin.16+ is shipped with ERR.USR being run as the Startup file in the Parm file, you should have it in memory already. If you changed the Parms to run some other USR function, type "-ERR.USR" from the main menu or the command box to activate its magic. This MUST be performed before any assemblies are performed. Failure here embeds a pair of zeros in your code, causing spectacular crashes.

2. Copy both of the error handling routines (CheckDiskError and CheckToolError) to somewhere within your program. You only need this handler at one point in the whole program.

3. Copy the macros into the start of each segment of the code, unless they are Put file segments. In that case, only copy the macros into the first (main) segment. Be sure you get all of the macros, including the KBD statement.

4. Finally, copy the ToolError and DiskError subroutines into each of the segments of code that you put the macros in. You'll probably want these to go somewhere near the end of each listing. Be sure to embed the correct filename in each of the STR statements there.

5. Change line 428 to point to your system shut down entry point so that the dialog box can have somewhere to jump to.

6. Now, everywhere you want to check for errors, use the following syntax:

```
          _InitCursor             ;For
Example...
          CheckToolError
     or:
          _Close CloseParms       ;For
Example...
          CheckDiskError
```

One last note: If you write a program using Skeleton, you don't need to do any of that (except steps 1 and 5). It's already set up and in place. Check it out.

WORKING WITHOUT A QUIT:

If you're working in an environment where you can't use a JMP to your system shutdown and quit (like writing an NDA), then re-write the CheckToolError Macro to read like this:

```
     FlagYN   KBD  "Assemble with line
numbers embedded? (1=Yes/0=No)"

     CheckToolError MAC
              DO   FlagYN      ;Do if
FLAG = 'Yes'
              USR
              BRK
              BRK
              FIN
              EOM
```

This then crashes to the monitor with A equal to the error number, Y equal to the Put file line number (or 0 if there is none), and X equal to the primary line number. With this way of doing it, you don't need any code in your program to handle the error except this macro. This technique doesn't look half as pretty, but it DOES do the job very well.

WORKING MULTIPLE LINK FILES:

When you're writing a program with multiple LNK files, there's one trick to keep in mind. Merlin.16+ command files are rapidly becoming far more sophisticated that just typing in a list of names.

Remember the KBD statement, that asks you, every time you assemble your file, if you want the line numbers embedded? Well, with link files, that's going to happen over and over...and over.

The trick? Put just the KBD line in your command file. Now Merlin only asks you once, and ignores all the other places where that line is in your listings. This speeds up assemblies a lot.

ANOTHER USEFUL THOUGHT

ERR.USR went through many variations during the 3 months I've been playing with it. One permutation, which some of you might want to consider because it's easier to use in most situations, is to combine the CheckToolError macro with the Tool Macro. Change it on your disk-- it's in the UTIL.MACS file. The advantage to this technique would be that every single tool call you made would automatically check for and handle your problems.

```
            Tool    MAC
                    LDX   #]1
                    JSL   $E10000
                    DO     FlagYN      ;Do
if FLAG = 'Yes'
                    USR
                    JSL    ToolError
                    FIN
                    <<<
```

The disadvantage (and the reason I have CheckToolError separate) is that you can't handle errors that you know are going to happen, but wish to ignore. (As an example, look at the new Generic.Startup listing. After starting up both the Event Manager and Quickdraw II Ross lets by the error about the tools already being started up! You'd never get past this!)

HOW DOES IT WORK?

Remember how we want X to be set to the current line number so we can display it? Well, that's exactly what ERR.USR does. It looks at zero page locations $D6-$D7 where Merlin.16+ keeps the current line number. Then it embeds an $A2 (LDX) and the actual two-byte line number.

Next, ERR.USR looks at the PutFlag ($06). If the flag is set, ERR.USR recognizes your file as a PUT file. It embeds a $A0 (LDY) and the two byte line number within the PUT file pointer ($D4-$D5). If the PutFlag is clear, then it embeds LDY and two 0's.

That's all there is to it.

Now, when an error happens, your code jumps to Tool Error and Disk Error with the error in A, the primary line number in X, and the secondary line number in Y. This short subroutine just checks the carry to be sure there IS an error. If not, it exits, and your program is none the wiser.

If there IS a problem, ToolError JSL's to the subroutine that really does the dirty work. Why does it JSL, even though we know we aren't coming back this way in this lifetime? Because we want (on the stack) the address of the string in the STR statement that follows it. This way, by just looking at the last two bytes on the stack, we can even have our dialog tell us what Source Code the error is in BY NAME!

ShowToolError gets entered with information everywhere. It converts the line numbers in X and Y to decimal (replacing the Put line number with "MAIN" if set to 0). It converts the error number in A to ASCII hex, so it can send each character to the screen. And it grabs the last two bytes off the stack and stores them into the dialog box data parameters so that the text in our STR statement gets displayed in the dialog box.

Then ShowToolError checks the environment to see if the Dialog Toolbox has started up. If not, it uses the text screen and the Tool Locator Toolkit to put up a box with the information in it. If so, it puts up a Stop Alert dialog box to tell you where you failed.

Then clicking on OK (or pressing Return) shuts down all of your tools and exits via the system QUIT handler (or even back to Merlin.16+ if you launched with the new '=' commmand!).

Back to the drawing board, as they say. But at least, this time, you know exactly where your problem is. Thanks to ERR.USR!

Editor: Jeff tells me that, like me, he accompanied his school-teacher wife to a remote and beautiful area, the Yosemite National Park. While living there for five years, he claims to have, "...fiddled to my heart's content, learning assembly language for fun and profit. I found the first immediately, and the second came last August when I landed a job as Technical Support for Roger Wagner Publishing."

*I'd like to make it clear that **my** wife accompanied **me** to the Bering Strait. It was only after dragging her out there that I said, "Okay, honey, I'm gonna quit my job and start writing newsletters."*

I just wanted to set the record straight. And if you read this, Tammy, they never believe me, anyway.

The Applesoft Connection, Part IV
A USR-Friendly Function

By Jerry Kindall

SnailMail: 2612 Queensway Drive
Grove City, OH 43123

GEnie: A2.JERRY
AppleLink PE: A2 Jerry
Internet: JerryK@cup.portal.com

Introduction

In previous installments of this series, I've covered most of what there is to know about passing parameters to and from Applesoft within your assembly language programs. There's still at least one area left for us to explore, though, and that's what I want to get into this time. I want to look at the seriously underutilized Applesoft USR function and see what it can do for us.

We've seen how the ampersand command is a user-defined command (maybe "programmer-defined" is a better term). USR is a user-defined function. If the difference between commands and functions isn't clear, consider the differences between the HOME command and the PEEK function. HOME is a verb; it tells the computer to do something. PEEK is closer to an operation than a verb. You can't just use the word PEEK in a program like you'd use HOME, you have to assign a PEEKed value to a variable, or print it. You use PEEK in an arithmetic expression; HOME is a command. That's the difference between commands and functions.

The USR Function

A typical USR function looks something like this:

```
         Y = USR (X)
 or:     PRINT USR (X)
```

You'll notice that like other functions such as PEEK, USR accepts (in fact, it requires) a parameter in parentheses after the word USR. What you put in the parentheses can be as simple as a numeric variable or a constant, or as complicated as any Applesoft numeric expression. The following statements are all completely legal:

```
         Y = USR (0)
         Y = USR ( PEEK (6502))
         Y = USR ( (K ^ 3) + 2 * V / 55.5 + PI)
         Y = USR ( Q + USR (V))
```

Note that in the last example, the USR function is used as part of a parameter to itself! This is perfectly legal; when you understand how Applesoft does its thing, you'll understand why this works.

Similarly, USR can be a part of any legitimate Applesoft expression. In other words, not only can you put a complex expression inside the parentheses, you can build one around it, too, like the following:

```
Y = USR (0) + 3
Y = USR ( PEEK (6502)) * PEEK (6503) / 256
Y = USR (1) + (USR (2) - USR (3))
```

Again, in the last example, you can see how you can have multiple USR functions in the same expression. The only limit is that you can have only 16 sets of parentheses open at one time (otherwise, Applesoft will give you a ?FORMULA TOO COMPLEX error). Other than that, USR behaves exactly like any other function.

The reason I'm going to the trouble of showing you all this (it doesn't really have much to do with showing you how to use USR in your assembly language programs, does it?) is to give you a feel for what kind of power USR gives you. Applesoft handles all the dirty work of evaluating the expressions for you. Because of this flexibility and its almost ludicrous ease of use, USR is ideal for machine-language routines which return a single numeric value.

Using USR

Installing a USR routine is similar to installing an ampersand routine. It involves getting an area of memory for the code, relocating the program if necessary, and setting up the USR vector. I won't cover this again here; for more information, see Part 3 of The Applesoft Connection (published in the April, 1989 issue of *The Sourceror's Apprentice*). The only difference is that instead of connecting your routine to the ampersand vector at $3F5-$3F7, you need to connect it to the USR vector at $0A-$0C. To keep things simple in this article, all the USR routines in the next few pages run at $300.

When your USR routine receives control, its argument (the expression in parentheses) will be in the Floating Point Accumulator (FAC) at $9D-$A3. Applesoft automatically places it there for you; there is no need to call FRMNUM ($DD67). You can start your USR routine by calling GETADR ($E752) to convert the floating-point number to a usable form (see part 1 of The Applesoft Connection in the January, 1989 issue of *The Sourceror's Apprentice*). You can also start out with a call to QINT to convert the FAC to a 3-byte integer (see part 2 of The Applesoft Connection in last March's issue of *The Sourceror's Apprentice*).

Applesoft always requires some sort of parameter with the USR instruction. If your routine doesn't need an input parameter, you can just ignore what's in the FAC, but whoever uses the USR function will need to use a parameter anyway. (Usually USR (0) is used in such cases -- the value inside the parentheses won't make any difference.)

When your routine is finished executing, put your numeric result (if any) into the FAC using SNGFLT, FLOAT, GIVAYF, or the unsigned-integer GIVAYF clone. (See part 1 of this series; also see part 2 if you want to pass a 3-byte integer back to Applesoft.) After you have done this, you can simply RTS back to BASIC. There's no need to find the variable and move the data to it; all you need to do is put your result into the FAC and return.

Like I said, Applesoft itself handles all the dirty work for you. Nice, isn't it?

An INKEY Function

Let's take a look at a simple USR routine. If you've used any other BASICs besides Applesoft, particularly Microsoft, you've probably seen INKEY$. INKEY$ is a special string variable which is usually null, except when a key is pressed. This allows you to easily check for a keypress without stopping the program. The Commodore 64's GET works similarly (it's different from Applesoft's GET).

Our sample USR routine has a similar function. USR (0) will always return 0, except when a key has been pressed, in which case the ASCII value of the key pressed is returned. The USR routine will always clear the keyboard strobe after reading a character, so that once a character is read, it won't be read again. The actual USR routine (lines 25-33) is ridiculously simple.

```
 1  * The Applesoft Connection
 2  * INKEY - A USR Function
 3  *
 4  * by Jerry Kindall -- August, 1989
 5
 6           ORG  $300
 7
 8  USRVECT  =    $0A
 9  KEYBD    =    $C000          ;keyboard input
10  STROBE   =    $C010          ;keyboard strobe
11  SNGFLT   =    $E301          ;float a 1-byte value into FAC
12
13  * Hook up INKEY routine
14
15  INSTALL  LDA  #$4C           ;JMP
16           STA  USRVECT
17           LDA  #INKEY
18           STA  USRVECT+1
19           LDA  #/INKEY
20           STA  USRVECT+2
21           RTS
22
23  * The actual INKEY routine
24
25  INKEY    LDA  KEYBD
26           BPL  NOKEY          ;no key pressed
27           AND  #$7F           ;convert to lo ASCII
28           BIT  STROBE         ;clear the keyboard
29           TAY
30           JMP  SNGFLT         ;return the key value
31
```

FAC, and exit by converting it to floating point format.

This can be a handy little routine to have in your arsenal, when you need it. Not only is the syntax simpler (X = USR (Y) vs. that horrendous formula), it's also quite a bit faster than two PEEKs, a multiplication, and an addition.

With a little effort you should be able to dream up all sorts of short, simple, and useful USR routines. For example, how about a USR function that, given a number, returns the address of that line in the current Applesoft program? It's quite simple as long as you know about FNDLIN ($D61A), which expects a line number in LINNUM and returns the address of the corresponding line in LOWTR ($9B). But I'll leave that, as they say, as an exercise for the reader.

Passing Strings

USR will also accept a string argument. For this reason, we probably should have included a JSR CHKNUM ($DD6A) at the beginning of our PEEK2 routine. (We don't care what we're passing our INKEY routine, so it wouldn't really be needed there.)

Although Applesoft doesn't check to see if the argument is a number or a string before calling your routine, it does make sure your result is numeric after your routine has finished. So if you accept a string parameter, you have to make sure that VALTYP ($11) contains a zero before your USR routine exits.

As it turns out, there's a routine called GETSTR at $E6DC which calls FRESTR (thus making sure the argument we received is a string and freeing the temporary string pointer), stores a zero in VALTYP, and puts the string length into the Y register. I'd suggest always using GETSTR within USR routines when you're accepting a string.

Here's a simple example. The USR routine below returns a one-byte "checksum" of all the characters in a string. The number can be used for error detection, or for indexing into a hash table. The checksum can detect transpositions as well as missing, added, or incorrect characters. Any two arbitrary strings have less than a 1/2% chance of producing the same checksum.

```
* The Applesoft Connection
* CHKSUM - A USR Function
*
* by Jerry Kindall -- August, 1989

         ORG  $300

USRVECT  =    $0A
SPTR     =    $5E          ;string pointer
FRESTR   =    $E6DC        ;see text above for info
SNGFLT   =    $E301        ;conv a byte to floating

* Hook up CHKSUM routine

INSTALL  LDA  #$4C         ;JMP
         STA  USRVECT
```

```
NOKEY    LDY  #0              ;return 0
         JMP  SNGFLT
```

A Two-Byte PEEK Function

How many times have you needed to PEEK a two-byte value from an Applesoft program? Most BASIC programmers resort to a clumsy two-byte PEEK formula that goes something like NUM = PEEK (ADR) + PEEK (ADR) * 256. Some programmers use DEF FN to set up a user-defined function that formula in programs that use it a lot. But, since we're studying the USR function, we'll write a USR routine to do the same thing. In this one, not only do we pass a value back, we get a value from the executing Applesoft program.

```
* The Applesoft Connection
* PEEK2 - A USR Function
*
* by Jerry Kindall -- August, 1989

         ORG  $300

USRVECT  =    $0A
LINNUM   =    $50             ;holds integer of FAC
FAC      =    $9D             ;floating point accumulator
GETADR   =    $E752           ;convert fac to integer
FLO2     =    $EBA0           ;convert integer to floating

* Hook up PEEK2 routine

INSTALL  LDA  #$4C            ;JMP
         STA  USRVECT
         LDA  #PEEK2
         STA  USRVECT+1
         LDA  #/PEEK2
         STA  USRVECT+2
         RTS

* The actual PEEK2 routine

PEEK2    JSR  GETADR          ;make it integer
         LDY  #0
         LDA  (LINNUM),Y      ;get low byte
         STA  FAC+2
         INY
         LDA  (LINNUM),Y      ;get hi byte
         STA  FAC+1
         SEC                  ;float it
         LDX  #$90
         JMP  FLO2
```

Once again, the PEEK2 routine is sweet and simple. We start off with a call to GETADR to convert the contents of the FAC to an integer value, in LINNUM. Then, using indirect indexed addressing, we get the two bytes starting at the specified address, stuff it in the

```
          LDA  #CHKSUM
          STA  USRVECT+1
          LDA  #/CHKSUM
          STA  USRVECT+2
          RTS

* The CHKSUM routine in the flesh

CHKSUM    JSR  FRESTR
          LDA  #0             ;init checksum
          CPY  #0             ;check length
          BEQ  EXIT           ;it's zero

          DEY
LOOP      CLC
          ADC  (SPTR),Y
          ASL                 ;shift left for next char
          ADC  #0             ;pick up overflow bit
          DEY
          CPY  #$FF           ;compare to length
          BNE  LOOP

EXIT      TAY                 ;pass our number back
          JMP  SNGFLT
```

Pass the Parameters, Please

USR's capabilities are pretty hip *(editor: Jerry must've been hanging out with Jay Jennings at the A2-Central Conference)*, but being able to pass only one parameter in each direction can sometimes be limiting. With a little work, though, you can pass even more parameters to and from your USR routines, although it won't be automatic.

When your USR routine gets control, TXTPTR (Applesoft's program counter) will be pointing to the character following the closing parenthesis. All you need to do is call the appropriate routines discussed in the first three parts of this series to parse these parameters.

I'd suggest enclosing the additional parameters in parentheses, using CHKOPN ($DEBB), CHKCLS ($DEB8), and CHKCOM ($DEBE) to keep Applesoft from becoming confused when trying to evaluate these additional expressions. (For example, X = USR (3), 2 + 3 isn't as clear as X = USR (3), (2) + 3 or X = USR (3), (2 + 3).)

Here's an example: A substring search routine might be called with a statement like X = USR (1),(A$,B$), where 1 would indicate the starting character position of the search, A$ would be the string to search, and B$ would be the string to search for. The answer would be passed back to X. And, of course, because it's a function, you could include any kind of mathematical manipulations you wanted in the same expression with the USR call. For many types of routines, it's more elegant than using the ampersand or CALL.

Peaceful Coexistence?

In Part 3 of this series, I described the de-facto protocol for writing ampersand routines so they'd be compatible with any other ampersand routines which follow the same rules. Unfortunately, no such standard has arisen for USR routines. What this means is that most USR routines "take over" the USR vector and assume that they are the only USR routine in memory. They make no attempt to determine if the USR call is for them, nor do they pass on any calls to previously-installed ampersand routines. I must admit that I've been guilty of this practice myself (the MicroDot GARB module is a USR routine). For the most part, this practice has caused no problems because hardly anyone uses USR.

Unfortunately, there's no logical or elegant way to allow USR routines to coexist in memory without conflicts. It's easy enough to relocate your USR routine in high memory, and also easy to save the previous USR vector, but how can you tell whether or not a particular call is for your routine, so you can pass it on if it isn't?

One way might be to check the value passed to the routine. Each routine would have a number associated with it: USR (1) would call one routine, USR (2) a different one, and so on. Unfortunately, this is not very mnemonic, and it means that any additional parameters must be passed after the USR function itself, as described in the section above.

Another way would be to check the character or token after the USR function, then call CHRGET to skip over it. For example, USR (X) INPUT would call one routine, and USR (X) PRINT would call another. This would work, but it's definitely not elegant.

There doesn't seem to be an easy solution to this problem, and that's undoubtedly one of the reasons that the ampersand has received so much glory while USR has languished in obscurity.

The End for Now

This article is the last installment of The Applesoft Connection. I've had a good time writing this series, and I hope it's been of benefit to some of you. Who knows, maybe someday I'll get an idea for another Applesoft Connection article and add a fifth part to the series. And of course, I plan to continue submitting articles to *The Sourceror's Apprentice*, so I won't vanish entirely.

Until we meet again, may your power supply run cool and your disk drives recalibrate quietly!

Ariel Publishing

Box 398
Pateros, WA 98846

509/ 923-2025

The Sourceror's Apprentice

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

Subscription prices in US dollars *(Canadian and Mexican subscribers add $5 per year, all other non-North American subscribers add $15 per year)*:

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

Ross W. Lambert.... Editor & Publisher
Tamara Lambert..... Subscriptions
Jay Jennings, Eric Mueller, Robert Moore, Jerry Kindall ...Contributing Editors
Rebecca Lambert... Stamp licking

Please direct all correspondence to:

Ariel Publishing
P.O. Box 398
Pateros, WA 98846

509/ 923-2025

The Sourceror's Apprentice is a product of the United States of America.

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