The Sourceror's Apprentice

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

Reality is for Non-Programmers

Well, here we are with the first REAL issue of *The Apprentice*. Unbeknownst to most of you, Volume 0 Number 0 was really an advertisement. Sure, it looked and felt pretty realistic, but it's true mission was to tell you all about our new newsletter. With this issue we begin your subscriptions and commence with the nitty gritty.

By way of announcement: we are going to offer a quarterly disk program so you can receive all the source code we printed in the newsletter already keyed-in and waiting to go (we'll throw in a few other ditties and oddities, too). It can save scads of time and will set you back a mere $20 per year. Just call or write... our address and other info is in the boilerplate on the back.

By way of confession: it is a little scary taking subscriptions from some of you. Recently I've received orders from people I've been reading and admiring for years. In talking with these members of the "illuminati", though, I discovered a common thread - they all wanted to lend their support to an all-assembly language, all-Apple II publication like this one. I think there is a place for *The Apprentice* in the Apple world, and I am proud to play a part in making it happen.

I am really only playing a part, though. Although I consider myself a pretty fair assembly language programmer, I can't hold a candle to some of you. On top of that, the playing field has about quadrupled in size over the last couple of years. The 65816 has scads of opcodes and addressing modes the simple little 6502 never dreamed of (and a precious few programmers know much about). On top of that, the GS toolboxes are an entirely new game, not to mention GS/OS. In short, the days are gone when one person could be the guru and dispense all manner of wisdom in all things assembly. We need each other, and I invite any and all to consider sharing some of your best "stuff" with the rest of the Apple II programming community. I have uploaded our writer's guidelines to GEnie's A2PRO library so you can download them or call or write and get 'em straight from us. Oh yeah, we usually pay between $50 - $75 per article, too.

And by way of correction: in spite of monumental effort, Mohawk Man and I let something slip through the cracks in our "perfect" GS start-up routine in Volume 0, Number 0. If you check out the error handling routine, it makes a toolbox call named ~Hexit. This little beastie is a member of the Integer Math toolset, which we neglected to start up. In my own defense, I DID spot this error before press time and fixed it - but then promptly typeset the wrong version of the source code. Good Lord, it's hard to get good help these days...

The error is not really noticeable because the Integer Toolbox is started automatically when the Tool Locator starts business. But Apple, Inc. says to start it up, so we shall. Somewhere in the midst of the ROM based toolset startups, just add a line like this:

```
~IMStartUp
```

That is all there is to it. The call does not affect the stack nor does it require any direct page space.

If all our mistakes continue to be that easy to fix, we'll be in terrific shape.

On a final 8-bit sort of note... I've received LOTS of mail and phone calls from people who have asked me not to forget "...those of us who still program the 8-bit machines..." Not to worry. I haven't and I won't. For one thing, I think the Apple IIc+ has extended the lifespan of 8-bit Apples (and increased the quality of that life, too). For another, I'm not entirely certain 8-bit computers will die anytime soon, anyway. The American school system is a LARGE market that moves slowly. There is and will continue to be a large demand for quality 8-bit software there and in other places. I plan to run at least one 8-bit article every issue, and there will usually be more.

With all that said, I now humbly request your attention to this month's feature presentations... (none of which I wrote - geez, I am one hardnosed editor!)

ACE is the Place...

by Eric Mueller

The ACE toolset, introduced with System Disk 3.2, is a very complete, very convenient way for you to compress digitized sounds down to half their original size or less. Interestingly enough, the technical notes released with System Disk 4.0 (GS/OS) did not include the ACE information, even though it did contain other release notes from the 3.2 package. This article is adapted from those not-too-well-distributed notes.

The ACE package (toolset $1D) consists of four major calls and several minor calls. The major functions are ACECompress, ACEExpand, ACECompBegin, and ACEExpBegin. The minor functions include such toolbox standards as ACEInfo, ACEReset, ACEStartUp, and so on. The heart of the ACE toolset is the ADPCM compression algorithm, formally known as Adaptive Differential Pulse Code Modulation. This algorithm can compress data in an 8:4 ratio (exactly one half the size of your original) or an 8:3 ratio. What that boils down to is, ADPCM is a very efficient method of packing bits, if you don't mind the fact that the expanded data will not be exactly like the original data.

ADPCM works by assuming that the data you're compressing--a sound wave sample--is relatively smooth, and continuous. Any sudden spikes in the noise, for example, will not compress well with ADPCM, if at all. ADPCM works by examining the difference between the previous byte of sound data, and the current byte. Imagine a simple ramp tone, for example. ADPCM will see very little difference between a sample byte and the previous byte--perhaps one or two units more, at the most. This data--the difference from one byte to another--is what is stored in the four or three bits that your sound is compressed to. If your sound has a sudden dropout, for example, where the value drops to zero for a byte or two, ADPCM can only indicate a rapid drop in its four (or three) bits. As you can see, four bits (versus three bits) allows greater variation in the changes from one byte to another in your sound, at the expense of slightly larger output.

There is an advantage to this, however. If your sound sample does include a dropout or spike, the expanded data will show it as being much less than it was, due to the nature of the compression algorithm. In addition, the ACE toolset boasts extremely fast data compression. This is because the ADPCM method requires one pass through the sound data, versus two or more passes for other compression algorithms (such as Huffman, popular with several disk packers). And, with ACE, you tell the toolset how long you want your compressed data to be--not the other way around!

ACE, however, cannot guarantee that the expanded data will be bit-for-bit the same as the original, uncompressed sound. As mentioned, this is a double edged sword, occassionally advantageous, occassionally problematic.

Enough with theory. In order to operate the ACER toolset and successfully compress and expand sound files, you need to follow several steps. I will present both a compression code fragment and expansion fragment. Both code fragments were adapted from Joe Jaworski's ACER utility (see details elsewhere in this issue). To use these fragments, you need to be certain that the host program has the integer math toolset started up, as well as the ACE toolset (of course).

When compressing a sound, there are three distinct steps. The first is to compute the buffer space needed for the compressed data. The second, to request this buffer are from the memory manager. Finally, the actual _ACECompress call is executed. Before the fragment can do anything, however, data space must be defined:

```
 * Equates that you should set or fill in:

method   dw   0          ;ACE compression method: 1 (8:4) or 2 (8:3)
ProgID   dw   0          ;program's ID from _MMStartup
```

```
NBlks      dw     0          ;number of 512 byte blocks in original data
OrigHandle adrl 0            ;a handle to the original sound data area

* Data area filled in by compression code

CompSize adrl  0             ;size of compressed data
CompHandle adrl 0            ;a handle to compressed data area
```

Next, the buffer space required for the compressed data must be calculated. According to the ACE release notes (page 16), the formula is:

```
 Bytes = NBlks * 64 * (5-method)
```

where NBlks is the number of 512 byte blocks in the original file, and method indicates which ACE method we're going to compress the data with. If method is one, then the data will be compressed with an 8:4 ratio, while method set to two indicates to ACE to use an 8:3 ratio. The easiest way to get the NBlks value is to use the GS/OS _GetFileInfo call, and take the block size (since a standard ProDOS disk block is 512 bytes).

```
         PushLong #0      ;push space for _NewHandle result now
         PushLong #0      ;push space for _Multiply result
         lda    #5        ;first, subtract the method from five
         sec
         sbc    method
         asl              ;now, multiply (*2)
         asl              ;(*4)
         asl              ;(*8)
         asl              ;(*16)
         asl              ;(*32)
         asl              ;(*64)
         pha              ;stick on the stack for _Multiply
         PushWord NBlks   ;push the number of 512 byte blocks
         _Multiply        ;and multiply the two together
         PullLong CompSize ;get the size necessary
```

The next step in the process is to allocate space for this buffer. You may be asking yourself why I pulled the CompSize value right off the stack and then just pushed it right back on. The reason is not that this fragment uses CompSize again (it doesn't), but that your host program is going to need where the compressed data is stored, in order to play it back or save it to disk!

```
         PushLong CompSize   ;specify how much space needed
         PushWord ProgID     ;host program ID (assigned by _MMStartup)
         PushWord #%1000_0000_0000_0100     ;locked and page aligned
         PushLong #0          ;(not used)
         _NewHandle           ;and allocate some memory
                              ;(here's where you do error checking)
         PullLong CompHandle  ;and get handle to work space
```

Now that I've got the size of the compressed data area computed and allocated, the compression can begin. Before setting up for the _ACECompress call, the _ACECompBegin resets internal settings in the ACE toolset. This call must be made at the beginning of each sequence of _ACECompress calls. Note that the _ACECompress call can be anywhere from instantaneous to several seconds long, depending on the size of the sound data, so I suggest you put up a dialog informing the user of the delay, or issue a conventional _WaitCursor call (changing the cursor to a wristwatch).

more ACE—

```
_ACECompBegin    ;must be made
PushLong OrigHandle ;push handle to original sound data
PushLong #0      ;offset (past above value) to sound data
PushLong CompHandle ;push handle to comp. (target) buffer
PushLong #0      ;offset (past ^ value) to start storing data
PushWord NBlks   ;push the # of 512 byte blocks to compress
PushWord method  ;push the compression method to use (1 or 2)
_ACECompress     ;and compress the data
                 ;(handle any kind of ACE error here)
```

That's all there is to it. In order to work with your data (and save it to disk, for example), CompSize is the number of bytes of compressed data, and CompHandle is a handle to the area where the compressed data is stored. Also, should you wish, you can compress the sound data 'on top of itself' by specifying the CompHandle equal to the OrigHandle. While this will save a few steps, you will also lose the original, since the compressed data will overwrite it.

As you can see, data compression with the ACE toolset is pretty elementary. Expanding a compressed file is just a hair more complex. The same three steps are involved--compute buffer size, allocate it, and expand--but the first one's a real doozy. Again, before I get started, a data area must be defined:

```
* Equates that you should set or fill in:

method     dw       0         ;ACE method: 1 (8:4) or 2 (8:3)
ProgID     dw       0         ;program's ID from _MMStartup
CompHandle adrl     0         ;a handle to the comp. sound data area
CompSize   adrl     0         ;size of compressed data, in bytes

* Data area; can be left as-is if you prefer:

ExpSize    adrl     0         ;size of expanded data
ExpHandle  adrl     0         ;a handle to expanded data area
Scratch    adrl     0         ;scratch area
```

Be certain that you set method equal to what it was when the data was originally compressed, or else you'll get some strange results (i.e. garbage). *Editor: Herein lies the single biggest problem with using "foreign" data. Who knows how it was saved? As usual, a standard protocol can help. See the ACER Standards article elsewhere in this issue.*

Now my fragment must calculate the size of the buffer necessary for the expanded sound data, by using this formula: ExpBlocks = CompSize / (64 * (5-method)). Again, method is the ACE compression method, CompSize is the size of the compressed data (in bytes), and ExpBlocks is the number of 512 byte blocks that the expanded data will occupy.

```
PushLong #0        ;push space for _LongDivide remainder
PushLong #0        ;push space for _LongDivide quotient
PushLong CompSize  ;push size of compressed data (dividend)
pea      #0        ;high word of divisor is always zero
lda      #5        ;first, subtract the method from five
sec
sbc      method
asl                ;then, multiply (*2)
asl                ;(*4)
asl                ;(*8)
asl                 ;(*16)
asl                 ;(*32)
asl                 ;(*64)
pha                 ;low word of divisor goes on stack
_LongDivide         ;and do the long divide
PullLong ExpBlocks  ;get the quotient off the stack
pla                 ;and forget about the remainder
pla
PushLong #0         ;push space for _Multiply result
PushWord ExpBlocks  ;ExpBlocks * 512 = bytes needed
PushWord #512
_Multiply
PullLong ExpSize    ;amount of memory needed (in bytes)
```

Allocating space for the expanded data is unsophisticated, as shown by the next step:

```
PushLong ExpSize    ;stick it back on stack for _NewHandle
PushWord ProgID     ;program ID (assigned by _MMStartup)
PushWord #%1000_0000_0000_0100 ;locked and page aligned
PushLong #0         ;(not used)
_NewHandle          ;and allocate some memory
                    ;(here's where you do error checking)
PullLong ExpHandle  ;and get handle to the compressed data
```

And finally, we begin the expansion step. Once again, this can take some time, so I suggest letting the user know of the delay.

```
_ACEExpBegin        ;must be made at beginning of each series
PushLong CompHandle ;push handle to compressed sound data
PushLong #0         ;offset past above to sound data
PushLong ExpHandle  ;push handle to exp. (target) buffer
PushLong #0         ;offset (past ^) to start storing data
PushWord ExpBlocks  ;the # of 512 byte blocks to expand
PushWord method     ;push compression method (1 or 2)
_ACEExpand          ;and expand the data out
                    ;(handle any kind of ACE error here)
```

Upon completion of the above step, the expanded data is stored in memory (a handle to it is stored at ExpHandle), and the length of the expanded sound data is stored at ExpSize.

The ACE toolset only returns a few errors, as shown by the following table:

error / name	comment
$0000/ACENoError	no error has occurred (operation successful)
$1D01/ACEIsActive	returned after startup call; indicates that the ACE toolset was already started up
$1D02/ACEBadDP	you have passed ACE a bad direct page address
$1D03/ACENotActive	returned after shutdown call; indicates that the ACE toolset was already shut down
$1D04/ACENoSuchParam	you asked for an out-of-range info parameter

ACER cont.

$1D05/ACEBadMethod	you passed ACE an invalid compression method
$1D06/ACEBadSrc	you passed ACE a bad source address
$1D07/ACEBadDest	you passed ACE a bad destination address
$1D08/ACEDataOverlap	the area of memory you specified for the expanded data will overlap with the compressed data
$1DFF/ACENotImplimented	that call does not exist in the ACE toolset

Who Be the Standard Bearer?

by Joe Jaworski and Eric Mueller

The ACE toolset, introduced with System Disk 3.2, allows you to compress sound data down to at least half it's original size, if not more. However, Apple has not defined a standard for sound files compressed with the ACE toolset, since it is primarily intended for developers to use within their own commercial products.

With the introduction of Joe Jaworski's ACER utility, which allows you to compress sounds with the ACE toolset, we would like to set forth the following protocols and file standards for Original, Compressed, and Expanded sound files as processed by ACER. Certain sections of this protocol are already being used by several sound-related applications, and as much testing as possible has been performed to ensure backward compatability.

ACER is available on the large on-line services (GEnie, CompuServe) for just the cost of downloading. In addition, you may wish to check with your local user group for a copy. ACER is freeware.

Proposed File Formats

ORIGINAL sound files

* Original Digitized Sound (ODS Format) files consists of the binary ($06) filetype.

* The Aux field of an ODS file contains the playback rate per the standard DOC frequency formula. The value recorded in the auxtype can be 1 ($0001) through 999 ($03E7). If a sound applications program detects a $0000 value, it should warn the user that the speed value is invalid and assume an initial playback rate of 200. The upper 4 bits (Bits 12-15) of the auxtype must always be set to zero.

* Optionally, the sound application can give the user an alternate choice of 427 as a default playback speed. This is the standard rate (22Kbits/sec.) used for Macintosh converted sounds.

* The data within an ODS file contains the raw 8-bit representation of a digitized audio signal, where $FF represents the most positive sound peak and $01 represents the most negative sound peak. $80 represents the zero crossing or null sound value. ODS files are prohibited from containing any $00 bytes (per the IIGS Sound Manager DOC routines).

COMPRESSED sound files

* Compressed Digitized Sound (CDS Format) files have a filetype of $CD. This filetype is currently unused.

* The data within CDS files contain the data generated by an ACECompress toolset call representing the resulting bytes of a compressed

original file using any present or future ACE METHOD value. Currently, only METHOD 1 (8:4) and METHOD 2 (8:3) values are supported in the ACE toolset, version 1.0. There is no other information of any -ind within the data portion of a CDS file.

* The auxtype of the file contains the playback rate of the original file (per the standard DOC frequency formula) AND the compression method that was used to compress the file. The value recorded in the auxtype for speed can range from 1 ($0001) through 999 ($03E7). The compression method is recorded in the most significant bit (bit 15) where a 0 ($0xxx) represents 8:4 compression and a 1 ($8xxx) represents 8:3 compression. For example:

auxtype	Description
$0239	8:4, 569 ($239) playback speed
$8239	8:3, 569 ($239) playback speed
$8070	8:3, 112 ($70) playback speed

Bits 10-14 in the auxtype are reserved and must be set to zero. These bits will be used when new compression methods and options are introduced with future versions of the ACE toolset.

EXPANDED sound files

Expanded sound files are identical sounding to original files in every respect. Because of the design of the ACE toolset and its compression and expansion routines, however, the expanded data will not be a bit-for-bit duplicate of the original sound file.

Please foreword any questions or comments regarding this standard to Joe Jaworski (GEnie JVJAWORSKI, CompuServe 73307,310). Any questions about the sample code should be forwarded to Eric Mueller (GEnie A2PRO.ERIC). Thank you.

The Applesoft Connection - And More

by Jerry Kindall

Part of Applesoft's flexibility comes from an ability to extend the language with machine-language subroutines, usually via the CALL statement or the ampersand vector. One of the things such subroutines need to be able to do is to pass data to and from BASIC variables.

When I was developing MicroDot (a compact ampersand-driven replacement for BASIC.SYSTEM), I needed to learn how to make my machine language routines communicate with Applesoft. I hope that what I learned can help you make more effective ampersand routines.

CHRGET & CHRGOT

Two small but extremely important Applesoft subroutines reside in the Apple zero page. Actually, it's just one routine with two entry points. The first routine, CHRGET (address $B1), advances the Applesoft program counter (known as TXTPTR, at address $B8) and falls through into CHRGOT (address $B7), which gets one character from the program and sets several 6502 status flags depending on what class the character belongs to.

More Applesoft Xface

If the program counter is at the end of an Applesoft statement (that is, if a colon or end-of-line character has been read), the 6502's zero flag will be set on return from CHRGET/CHRGOT. If the carry flag is clear, the character read was a digit (zero through nine); other characters leave the carry flag set.

Syntax Checks

At a slightly higher level than CHRGET & CHRGOT are other routines which Applesoft uses internally to perform syntax checks. One routine that gets a lot of use in ampersand routines is CHKCOM, at $DEBE. This routine checks the current Applesoft program character (the one pointed to by TXTPTR) and makes sure that it is a comma. If a comma is not found, a ?SYNTAX ERROR is issued; if the comma is there, CHRGET is called to read the next character.

Closely related routines are CHKOPN ($DEBB) and CHKCLS ($DEB8), which check for opening and closing parentheses, respectively. CHKCOM, CHKOPN, and CHKCLS all fall through into a routine called SYNCHR, at $DEC0, which checks for the character in the Accumulator. Another useful routine is ISLETC ($E07D) which checks the character in the accumulator to see if it is a letter, and returns with the carry set if it is, and the carry clear if it is not.

Reading Applesoft's Mind

The routines we've looked at so far are quite useful in ampersand routines, but they won't help you get at Applesoft variables. To do that, you need different Applesoft routines. One of the most useful is GETBYT, at $E6F8. This routine evaluates the Applesoft expression at TXTPTR and returns a byte value from zero to 255 in the X register. If the value is outside that range, an ?ILLEGAL QUANTITY ERROR will be generated.

COMBYT, an almost identical routine, is at $E74C. COMBYT calls CHKCOM first and then jumps to GETBYT. It is functionally equivalent to JSR CHKCOM followed by JSR GETBYT, but it saves a statement. It is used quite frequently in ampersand routines.

If you're working with hi-res graphics, you might find HFNS ($F6B9) useful. This routine is used by Applesoft's HPLOT command to parse an X-Y coordinate pair. Upon return from this routine, the Accumulator contains the Y coordinate, and the X and Y registers contain the low and high bytes of the X coordinate, respectively. HFNS will automatically issue an ?ILLEGAL QUANTITY error if the coordinates are out of range, and it also checks for the comma between the numbers.

It is worth mentioning, just in case it's not obvious, that these routines evaluate not just numbers or variables, but expressions, such as X + 2, SIN (Y), and so on. These routines will accept any legal Applesoft numeric expression as long as it evaluates to a legal value (such as 0-255 for GETBYT).

FRMNUM and GETADR

FRMNUM ($DD67) is Applesoft's main numeric expression parser. It evaluates the expression pointed to by TXTPTR and leaves the result in floating-point format in the FAC, the Floating-point ACcumulator, at $9D-$A3. Unlike the 6502's Accumulator, the FAC is operated on not by hardware but by the floating-point math routines built into Applesoft.

For assembly-language programmers, the floating-point format is a little awkward to work with, unless you are planning to use Applesoft's math routines in your program. In any case, that's quite another subject and an article all its own. GETADR ($E752) is a routine that converts the FAC to a two-byte integer value stored in LINNUM ($50-$51), in the usual low-high format. GETADR accepts signed and unsigned integers from -65535 to 65535. (Other values will produce an ?ILLEGAL QUANTITY ERROR.)

Use FRMNUM and GETADR as a team. First call FRMNUM to get the value, then call GETADR to convert it to a form you can work with easily.

Mind Control

How do we go the opposite way, passing values back to an Applesoft program's variables? It's a little more complicated than what we've looked at so far. To simplify things a little at first, we will assume that you want to pass a value back into an Applesoft floating-point (real) variable.

Passing a value back to a real variable is a three-step process. First, find the memory location of the variable. Then convert the number to floating-point format. Finally, move the variable to its proper place as determined in step 1. Here are some Applesoft routines you will find helpful.

PTRGET ($DFE3): finds an Applesoft variable's memory address and puts a pointer to the variable in VARPNT ($83-$84). The variable's name is left in VARNAM ($81-$82). It will work with any type of variable, from an integer to an array element to a string. If the variable does not already exist, PTRGET will create it for you.

CHKNUM ($DD6A): verifies that the most recent variable found by PTRGET was a numeric variable. Since PTRGET can find a string variable, you should use CHKNUM to verify that the ampersand command has a legitimate numeric variable where one was supposed to be.

SNGFLT ($E301): converts the byte in the Y register to an unsigned floating-point number in the FAC.

FLOAT ($EB93): converts the byte in the Accumulator to a signed floating-point number in the FAC. (Negative numbers should be in two's-complement format.)

GIVAYF ($E2F2): converts the two-byte signed integer in the Accumulator (low byte) and the Y register (high byte) to its floating-point equivalent in the FAC. (As with FLOAT, negative numbers should be in two's-complement format.)

MOVMF ($EB2B): copy the floating-point number in the FAC to the address pointed to by the X register (low byte) and the Y register (high byte).

Integer Variables

Passing values back to integer variables is actually simpler than working with real values. All you need to do is find the variable, then move the value into the variable. There's just one pitfall: Applesoft integer variables are stored high-byte first, while 6502 integers are typically stored low-byte first. This means that you must reverse the order of the bytes if you're passing back a two-byte value. Remeber, also, that integer values have a range of -32768 to 32767, so if you put a large positive number into an integer variable, the value will turn out negative.

Unsigned Two-Byte Values?

You may have noticed that Applesoft does not have a built-in subroutine for converting an unsigned two-byte value to an equivalent value in the FAC. This turned out to be a major stumper when I was writing MicroDot, since I wanted to pass a file's auxtype value (which is two bytes in length) back to an Applesoft variable. If I used GIVAYF, any numbers over $7FFF hex would be returned as negative integers! I overcame this tricky spot after careful study of the Applesoft routines mentioned above.

To convert an unsigned two-byte value to floating-point format, you must first store the desired value at FAC+1 and FAC+2 ($9E and $9F), in the same backward order used with integer variables (high byte first). Then you must set the carry flag, load the X register with the number $90, and call FLO2 at $EBA0. This odd string of events will lead to the desired floating-point value being deposited in the FAC, ready to be moved out to a real variable.

Here's how it would look as a section of assembly-language code. The input parameters are the same as used with GIVAYF, and, in fact, this routine can be used as a direct substitute for GIVAYF.

```
FAC          =     $9D
FLO2         =     $EBA0

GIVAYF2      STY   FAC+1
             STA   FAC+2
             SEC
             LDX   #$90
             JMP   FLO2
```

Error Handling

All of the routines mentioned so far, since they are routines within Applesoft, can issue errors such as ?ILLEGAL QUANTITY ERROR or ?SYNTAX ERROR. In other words, Applesoft handles the

More Applesoft

errors for you. You do not have to deal with these errors because Applesoft will not return control to your program if one of these errors occurs. (The error will be printed on the screen, or passed to the ONERR GOTO routine, as appropriate.)

You can also voluntarily give up control to Applesoft's error handling if you discover something wrong with the command syntax by jumping to SYNERR ($DEC9) or ILQERR ($E199), which issue SYNTAX and ILLEGAL QUANTITY errors, respectively. If another error code would be more useful, you can load the X register with an Applesoft error code and jump to ERROR ($D412). Applesoft will take care of all the dirty work for you.

Putting It All Together

Here are a couple of routines similar to ones I used in MicroDot. GETNUM gets a number from an Applesoft expression; PUTNUM and PUTBYT return values to Applesoft variables. (PUTBYT returns a single byte while PUTNUM returns an unsigned two-byte value.)

There's a tricky part in lines 65 and 66. The PUTNUM/PUTBYT routine is designed to work with integer variables (if the number is greater than 32767, it will be negative when returned to an integer variable) or real variables (the number will come through unsigned).

Lines 65 and 66 take advantage of the fact that PTRGET puts the name of the variable into VARNAM. It is an Applesoft convention to use the high-order bits of the two-character variable name to denote the variable's type. If the variable is real, the high bits are both off (positive); if the variable is integer, the high bits are both on (negative). Since we have already established that the variable is numeric with a call to CHKNUM, we can simply test the first character of the variable name. If it's positive, we know it's real and so we bypass the integer-variable routine and go to the real-variable routine.

```
 1  * The Applesoft Connection
 2  * Passing Data Between BASIC & Machine Language
 3  *
 4  * Routines by Jerry Kindall
 5
 6  * Zero page locations:
 7
 8  LINNUM    =    $50       ;FAC converted to integer
 9  VARNAM    =    $81       ;variable name
10  VARPNT    =    $83       ;variable pointer
11  FAC       =    $9D       ;floating point accumulator
12
13  * Applesoft routines:
14
15  CHRGOT    =    $B7       ;get last character
16  FRMNUM    =    $DD67     ;get numeric formula
17  CHKCOM    =    $DEBE     ;syntax check for comma
18  PTRGET    =    $DFE3     ;get variable pointer
19  GETADR    =    $E752     ;convert FAC to integer
20  MOVMF     =    $EB2B     ;move FAC to variable
21  FLO2      =    $EBA0     ;convert integer to real
22
23  * GETNUM routine
24  *
25  * Gets a number into Acc (low) and X reg (high)
26  * If at end of Applesoft statement, return zero
27
28  GETNUM    JSR  CHRGOT    ;check current character
```

```
          BEQ  :ZERO     ;we are at end of statement

          JSR  CHKCOM    ;make sure we have a comma

          JSR  FRMNUM    ;get the number into FAC
          JSR  GETADR    ;convert FAC to integer

          LDA  LINNUM    ;get the low byte into Acc
          LDX  LINNUM+1  ;and the high byte into X

          RTS

:ZERO     LDA  #0        ;get the zero into Acc
          TAX            ;and to X also

          RTS

* BYTPUT / NUMPUT routines
*
* BYTPUT puts a single-byte value into a variable
* NUMPUT puts a two-byte unsigned integer into a variable
*
* On entry: X reg = high byte, Acc = low byte
* BYTPUT zeroes X register before proceeding

BYTPUT    LDX  #0

NUMPUT    PHA
          TXA
          PHA

          JSR  CHKCOM    ;check for required comma

          JSR  PTRGET    ;get address of variable
          JSR  CHKNUM    ;make sure it's numeric

          LDA  VARNAM    ;if first letter of name
          BPL  :REAL     ; is postive, it's real

:INTEGER  LDY  #0        ;store high byte in variable
          PLA
          STA  (VARPNT),Y

          INY            ;store low byte in variable
          PLA
          STA  (VARPNT),Y

          RTS

:REAL     PLA            ;store high byte in FAC
          STA  FAC+1

```

Applesoft...

```
          PLA              ;store low byte in FAC
          STA   FAC+2

          SEC              ;convert it to floating
          LDX   #$90
          JSR   FLO2

          LDX   VARPNT     ;move FAC to variable
          LDY   VARPNT+1
          JMP   MOVMF      ;move it out & return
```

Programming with Class 1

by Mohawk Man

Warning! Warning! The program associated with the following article is not pretty! It is not user friendly! And if you don't follow instructions, it could FORMAT or ERASE your hard drive! Phew, glad I got that off my chest.

When Apple released GS/OS last September, they made sure the ProDOS 16 programs in use would still work. There are two classes of MLI (Machine Language Interface) calls in GS/OS. The first are Class 0 calls, or the old ProDOS 16 calls. The new ones are strictly for use with GS/OS and are called Class 1, or GS/OS calls.

One of the nifty features of Class 1 calls is the way that some of them "take over" your system when you call them. By this, I mean they draw their own dialog boxes, complete with menus, buttons, etc. All you have to do is call them. This holds true whether you're using a text screen or a SHR screen. They adjust themselves accordingly.

We're going to program three different Class 1 calls this month. The first is transparent and the other two will pop up and 'take over'. We'll be using the text sceen for this to save on the amount of source code we need to type. Our objective is to erase or format a disk using GS/OS.

The first thing our program does is to start the needed tools. And the only reason these tools are started is to make Apple happy. And so that this program will run should they ever change the GS (don't hold your breath). Then the current text screen globals are saved and init'ed anew. And then we're into the program.

Shades of the II+! Yes, press your CAPS LOCK key down. To save on code (what? 3, 4 lines???) I just check for capital letters at the keypress place. When the program starts it looks for device number one and prints the device name to the screen if it finds it. Pressing the space bar will toggle through all the devices on line.

Time for a knowledge break, boys and girls. A device is a physical piece of equipment that transfers information to or from the Apple IIGS. This includes disk drives, printers, mice, the keyboard, the screen, and more. Instead of communicating with a device by slot and drive as in the days of old, GS/OS just deals with device names. It really doesn't care where the device is located. A GS/OS device name begins with a period and is 2 to 32 characters long.

Back to the program. While toggling through the devices, you can hit F to format the current device or E to erase it. Don't worry too much about erasing your hard drive without knowing it. When you hit E (or F) GS/OS takes over and puts a dialog box on your screen. At this point you have the option of specifying which FST (File System Translator) you want the disk erased (or formatted) with. You then press RETURN to proceed with the operation or hit ESCAPE to cancel. If you are using the Format option, you also get the chance to specify which interleave you want on your disk.

Just to be on the safe side, you may want to turn off your hard drive while you play with the program. *(Editor: or use someone else's computer.)*

If you try to Erase or Format a non-block device, the program will return an error. After all, if you Format your printer today, you may regret it tomorrow.

The program is fully commented, but I want to go over a couple things that had me stumped.

First, if you pop a disk out of a drive and then try to format or erase it, you'll get error $2F returned. You'll never find that error code in any book unless you have the "GS/OS Reference, Volume 1 Beta Draft" from APDA. This error translates to "device off-line or no media present".

Second, I had a tough time getting the DInfo call working because of preconceived notions about parameter blocks. GS/OS parameter blocks all start with a parameter count. This makes it easy to add parameters to GS/OS calls should Apple decide to do so in the future. All existing calls will continue to work as they should and new applications will be able to take advantage of any new features.

Starting the parameter block with a parameter count was not the stumbling block, though. The second parameter (after the count parameter) of DInfo is a long pointer to a result buffer where GS/OS returns the device name. This buffer is not just space for the name, as I first thought.

A GS/OS result buffer has three parts. First, a buffer length word that tells the total length of the buffer, including the buffer length word. Next is the string length word that specifies the total length of the string that follows. There are certain situations when GS/OS will return an error telling you that you have not set aside enough space for the string you want returned. You need to get the GS/OS Reference manual and read this section very carefully! I worked literally for hours without getting anywhere because I skimmed that part of the manual and didn't stop to understand it fully.

So, type in the program, run Macgen, and try out GS/OS Class 1 calls for yourself. Then go write some 16-bit code!

```
********************************
*          PunkWare            *
*          Presents            *
*                              *
* GS/OS Class 1 Call Examples  *
*                              *
* Another Mohawk Man Creation  *
*         Written for          *
* The Sourceror's Apprentice   *
********************************

         xc
         xc
         mx    %00          ;full 16-bit mode, thank you
         rel
         dsk   CLASS1.L
         lst   off
         exp   off
         use   class1.macs

         do    0            ;turn off the assembly for a sec...
GSOS     mac                ;define this macro
         jsl   $E100A8
         da    ]1
         adrl  ]2
         eom
         fin                ;and now turn assembly back on
```

Programming With Class 1

```

DInfo     =      $202C
EraseDisk =      $2025
FormatDisk =     $2024

          phk
          plb                ;make the program bank = data bank
          _TLStartUp         ;start this for nothing except Apple's
say-so
          ~MMStartUp         ;and this one, too
          PullWord UserID    ;save this for no good reason
          _TextStartUp       ;yeah, and this one <grumble grumble>
          _IMStartUp         ;gee, another???
          ~GetInGlobals      ;save globals on the stack
          ~GetOutGlobals     ;we'll pull them off when we're done
          ~GetErrGlobals
          ~InitTextDev #0    ;init the text input
          ~InitTextDev #1    ;init the text output

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

RadCode   ~TextWriteBlock #HipVerbage;#0;#HVEnd-HipVerbage

          inc   devNum       ;point to the next device
          jsr   DeviceInfo   ; and get the device name

          ~TextWriteBlock #devName;#4;devName+2

GetChar   ~ReadChar #0       ;get a char from the keyboard
          pla
          and   #$00ff       ;strip off the useless high byte
          cmp   #" "         ;was the space bar pressed?
          beq   RadCode      ;if so, look at the next device
          cmp   #"F"         ;was the F key pressed?
          beq   Format       ;if so, Format the current device
          cmp   #"E"         ;was the E key pressed?
          beq   Erase        ;if so, erase the current device
          cmp   #$9B         ;was ESCAPE pressed?
          bne   GetChar      ;if not, get another key from Mr User

:Quit     jmp   WeBeDone     ;ESC was pressed so we're outta here!

*==================================================
* format the current device

Format    GSOS  #FormatDisk;FormatParms
          bcc   :rts         ;Mr User pressed ESC from the dialog box
          beq   :rts         ; if the carry is set and A = 0
          jsr   Error
:rts      brl   RadCode

FormatParms da 3             ;number of parameters
          adrl  devName+2    ;device name to format
          adrl  NewName      ;new name for disk to be formatted
          ds    2            ;result shows which FST was used to format
```

adverse
effect
confusion

```

*==================================================
* erase the current device

Erase    GSOS   #EraseDisk;EraseParms
         bcc    :rts         ;Mr User pressed ESC from the dialog box
         beq    :rts         ; if the carry is set and A = 0
         jsr    Error
:rts     brl    RadCode

EraseParms da   3            ;number of parameters
         adrl   devName+2    ;name of device to erase
         adrl   NewName      ;new name for disk after erasure
         ds     2            ;result shows which FST was used in erasure

NewName  strl   '/BLANKDISK' ;very original new name for the disk

*==================================================
* get device info

DeviceInfo GSOS #DInfo;DIParms
         bcc    :1
         cmp    #$11         ;have we hit the end of the device list?
         bne    :Not11       ;if not, check for another error
         lda    #1           ;if so, start over with device #1
         sta    devNum       ; and
         bra    DeviceInfo   ; do the DInfo call again
:1       rts

:Not11   jsr    Error
         brl    RadCode

DIParms  da     2            ;let's just go for the minimum parms
devNum   da     0            ;which device to get info on
         adrl   devName      ;pointer to result space for device name

devName  da     35           ;total buffer length (weird GS/OS stuff!)
         ds     33           ;space for length word + device name

*--------------------------------------------------
* some hip verbage for people who read such stuff

HipVerbage hex 8c
         asc    "                            "
         asc    "GS/OS Class 1 Calls"
         hex    8d,8a,8a
         asc    "Hit spacebar to toggle the device. "
         asc    "Hit F to format, E to Erase, and ESC to quit."
         hex    8d,8a,8a
         asc    "Current device: "
HVEnd

```

Class 1

```
*=================================================

Error
         sta   TempA
         ~HexIt TempA     ;turn the error number into an integer
         PullLong ErrNum
         ~TextWriteBlock #ErrMsg;#0;#ErrEnd-ErrMsg
         ~TextWriteBlock #ErrMsg2;#0;#ErrEnd2-ErrMsg2
         ~ReadChar #0
         pla
         rts

ErrMsg   hex   8d,8a,8a,8a
         asc   "Error number #$"
ErrNum   ds    4
ErrEnd
ErrMsg2  asc   "       Press any key to continue..."
ErrEnd2
TempA    ds    2

*=================================================

WeBeDone                  ;restore everything we messed up
         _SetErrGlobals   ;and then get outta here
         _SetOutGlobals
         _SetInGlobals

         _IMShutDown
         _TextShutDown
         ~MMShutDown UserID
         _TLShutDown
         _QUIT QuitParms

UserID   ds    2

QuitParms adrl $0
         ds    2
```

Worldwide prices in US dollars effective January 1, 1989:
1 year...$28, 2 years...$52, 3 years...$72

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.