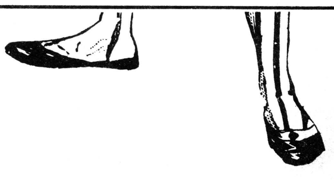
April, 1990
Volume 1, No. 2
Beware the April
Fool

8/16

The Journal of Apple II Programming

$3.50

Bert Kersey Where are you?



This month in *8/16*:

Nite Owl's

Slide-On
Slide-On
Slide-On
Slide-On™
Brand

Battery Replacement Kit for Apple IIGS Computer

- **Fantastic Savings**
- **Easy Installation**
- **No Solder Required**
- **Complete Instructions**
- **10 Year Shelf Life**
- **Top Quality Lithium**

Patent Pending

New kit restores your Apple IIGS and saves you the hassle and expense of normal solder type batteries.

If you purchased an Apple IIGS computer before August 1989 (512K model), a Lithium battery was soldered onto the computer board at the factory and the internal clock started ticking. It is just a matter of time until the battery runs out of juice and your computer forgets what day it is and any special settings you have selected in the Control Panel.

If the software you are running uses the date and time to keep track of records you could be in for real trouble when the clock runs out. The IIGS is also known to lose disk drives along with numerous other side effects caused by a dead battery.

Before the introduction of Nite Owl's Slide-On battery, the normal method for replacing the IIGS battery was to pack your computer up and take it to your local Apple dealer. The service department would solder on a new one and charge you a small fee, usually between $40 and $80. That was very inconvenient, time consuming, and expensive for the typical computer owner.

Slide-On battery replacement is not much more difficult than changing a light bulb. Using wire cutters, scissors, or nail clippers, the old battery is removed leaving the original wires still soldered to the mother board. The new Slide-On battery has special terminals which have been designed to fit onto the old battery wires. It usually takes only a couple of minutes. Complete, easy-to-follow instructions are included with every kit.

Typically, our customers have reported that the original equipment batteries have an average life expectancy of 2 to 3 years. This is about half as long as they were supposed to last. Slide-On replacement kits include Heavy Duty batteries which should provide for a longer battery service life.

We highly recommend that every IIGS owner keep a spare battery on hand, ready for when the inevitable battery failure occurs. These Lithium batteries have a shelf life of over 10 years. The Slide-On kits come with a full 90 day satisfaction guarantee.

Purchase Slide-On battery kits from your local dealer, distributor, user's group, or direct from Nite Owl.
School Purchase Orders are welcome.
Order your IIGS a spare today!

Telephone:
(913) 362-9898
FAX:
(913) 362-5798

Photo-Copyable

Quantity	Pricing
1+	14.95
10+	12.00
50+	10.00
100+	9.00

Sales Tax
Kansas residents 6%

Shipping•Handling
Add $2.00 / Order
Overseas add $5.00

Nite Owl Productions
Slide-On Battery Dept. A
5734 Lamar Avenue
Mission, KS 66202
USA

(Cut & Paste Address Label)

Ship to:
____
____
____
____
Telephone #: ____________

• Bill To •
Cash, Check, Money Order
VISA
Master Card
Purchase Order

Credit Card or PO#	Expiration Date

Quantity	Description	Price	Amount
	Slide-On Battery Kits		
Signature for Credit Card Orders		Kansas Sales Tax	
I am interested in other batteries for:		Shipping & Handling	
		TOTAL	

Prices may Change without notice.

8/16

Copyright (C) 1990, Ariel Publishing, Most Rights Reserved

Publisher & Editor-in-Chief	Ross W. Lambert
Classic Apple Editor	Jerry Kindall
Apple IIgs Editor	Eric Mueller
Contributing Editors	Walter Torres-Hurt
	Mike Westerfield
	Steve Stephenson
	Jay Jennings
Subscription Services	Tamara Lambert
	Becky Milton

Introductory subscription prices in US dollars:

• *magazine*
1 year $29.95 2 years $56

• *disk*
1 year $69.95 6 mo $39.95 3 mo. $21

Canada and Mexico add $5 per year per product ordered. Non-North American orders add $15 per year per product ordered.

Direct all correspondence to: Ariel Publishing, Inc., P.O. Box 398, Pateros, WA 98846 (509) 923-2249.

The Publisher's Pen

by Ross W. Lambert

Though many have tried to legislate and regulate it into impotency, the Law of Supply and Demand is as immutable as gravity, and just as natural. It is the way things are, especially in the software and publishing businesses.

Case in point: the good folks at Softdisk™ Publishing are often the first to feel the winds of change because they distribute so *many* programs each month. They are fine tuned to both the consumer market *and* the labor market in the programming community.

Jay Wilbur, the Apple II editor at Softdisk, has been beating the bushes for weeks for 8 bit software for the tens of thousands of IIe, IIc, and IIc+ owners who subscribe. There is a definite shortage of new packages for that market, even though all those 8 bit computers out there are not being incinerated.

I find that interesting. He and I both agree that it has something to do with the fact that programmers are more "into" their machines than most folks, and so have moved up to the GS in disproportionate numbers.

This has produced the rather unique situation where, to attract 8 bit developers, Softdisk has increased their compensation for 8 bit programs to a level *above that for GS software!* Even El Casa Ariel (uh, that's us) is in dire need of 8 bit related articles and source code.

For those of you who can act quickly, this is a tremendous opportunity. I think it is fairly safe to say that The Law of Supply and Demand will come into play again as Softdisk is flooded with 8 bit material. But a free market system rewards the nimble. Give Jay a call (318) 221-5134.

On a larger scale, and this is really the point, don't forget that the 5 million Apple IIe's, IIc's, IIc+'s, and even II+'s did not mysteriously vaporize overnight. Claris and Beagle Bros. have been making a small fortune because they recognize this fact. == Ross ==

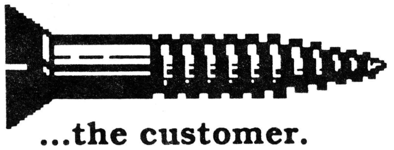
We tease those we love - this is an April Fool's Joke, John

Apple Discontinues Macintosh!

Cupertino - (APL) - Apple Computers, Inc. announced today that they are discontinuing production of the entire Macintosh™ line, effective January 1st, 1992. Company spokesman Jonathan (Darth) Vader read from a prepared statement which stated that, "It is the position of the board of directors that dropping the Macintosh line will maximize profits in the short run, thereby heading off the decline in the price of our stock. Having examined the recent earnings history of the company, the board came to the conclusion that Apple's neglect of the Apple II™ line over the last six years has resulted in billions of dollars of profits for the company, and with very little overhead. The board feels that a similar sort of neglect in the mutli-billion dollar Macintosh market should likewise produce as much or more immediate profits."

During the barrage of questions that followed the announcement, Vader was asked if Apple, Inc. was going to announce a new hardware product to replace the Macintosh and Apple II lines. He replied, "For the record, it is against company policy to discuss unannounced products. Off the record, however, I can tell you that we have a hip and happening little 1 gigabyte machine that we've tucked into a Pee-Chee. The CD ROM disks that drive the machine double as frisbees, so the $35,000 retail price ought to be well worth it."

Using a single hires graphic generated by an Apple II+, Vader also displayed the new Apple corporate logo:



...the customer.

Vader refused comment about reports that Joe Louis Gassey had been forced to resign during the recent upheavals at the company. Gassey, a flamboyant individualist who headed Apple Albania during the early 1980's, has been rumored to have filled out employment applications at Laser Computers, Microsoft, and Ariel Publishing.

Ariel's President, Ross W. Lambert, refused to confirm or deny the report, but noted that, "Ariel Publishing remains committed to programming in BASIC (among other things), and since Mr. Gassey has publicly stated that, 'BASIC is dangerous to the mind...', I have serious doubts as to his ability to prosper in our environment, not to mention pronounce the word 'dangerous'. Besides, there are no Albanian restaurants in Pateros, WA, and everybody here wears sweat pants and T-shirts to work. "

In a related development, Apple CEO Ron Skulby is reported to have recently signed a lucrative acting contract with Coca-Cola, Inc. The Coke company spokesperson stated that, "We have signed Mr. Skulby to be our new product spokesman during a multi-year commercial campaign."

When asked why he accepted the spokesperson's position with his former arch-rival, Mr. Skulby said, "Hey man, I need the money. The board here at Apple tied my salary to future profits."

In an interview with Bahbah Waters on national television, Italian computer industry analyst Tom Swiharti speculated that, "Apple's moves are well-calculated and bold. With no hardware to sell, they are a company without a product. This is a highly unusual situation, unqiue in the short history of the microcomputer industry. However, the total lack of overhead required to maintain this position might possibly allow Apple to improve the bottom line on less gross revenue. And if that fails, they can always sell linguini..."

Former Apple evangelist Guy Yamaha was also reported to have contacted Joe Louis Gassey about starting a new hardware company in partnership with 65816 producer William Mensch. In a memo recently leaked to the press, Gassey is reported to have replied, "Why de hack nut, mon, we both be out of a job now anyways. I be doin' it so long as I donna have to talk wit dat Mensh (sic) mon."

This development startled many in the industry, but as the saying goes, "Neccessity (and unemployment) makes for strange bedfellows."

Bascially Applesoft

Pretty Polygons

by Barry D. Hatchett

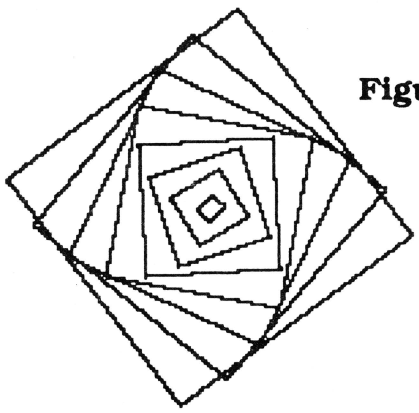


Figure 1

Applesoft has several built-in commands for hi-res graphics. Unfortunately, no commands are available for plotting polygons or circles. I have written a unversal polygon plotter which can draw regular polygons with any number of sides, and can also "rotate" them around the X, Y, and Z axes, allowing you to "distort" your polygons in all sorts of interesting ways. It uses Applesoft's built-in SIN and COS functions, but you don't have to understand trigonometry to use it.

The routine itself can be found in Listing 1. It's short and simple. I have attempted to keep it to only one or two statements per line, just to make it easier to read and understand. In situations where more speed is required, you could "optimize" it by putting more statements on a line.

To use the routine, you must first set up a few variables which serve as the parameters for drawing your polygon. First, set XC and YC to the X and Y coordinates of the center of your polygon. Set R to its radius — the length (in pixels) from the center to a vertex of the polygon. Set N to the number of sides in your polygon: 5 for a pentagon, 8 for an octagon, and so on. Circles can be drawn by using a large number of sides, say 100. Set XR, YR, and ZR to the rotation (in degrees) around the X, Y, and Z axes, respectively. (I will discuss the effects of the various rotations shortly.) Make sure you have initialized a hi-res display page and set the plotting color, then GOSUB 1000. Like magic, your polygon appears on the hi-res screen.

Internally, the routine uses a few variables of its own for scratch. These variables are X0, Y0, X, Y, P2, EX, EY, EZ, EV, S, E, and I. You don't have to set these, but don't try to use them in the program that calls the routine, because you will lose the data contained by them whenever you plot a polygon.

Rotation

As I said earlier, you can rotate your polygons around three axes. The X and Y axes are just as you'd expect. Rotating a polygon around the X axis will cause it to get shorter because you are "viewing" it from an angle. Rotating around the Y axis will cause it to get narrower. Both the X and the Y rotations can range from zero to 90 degrees. At zero degrees, the polygon is displayed normally. At 90 degrees, the polygon is displayed as a straignt line!

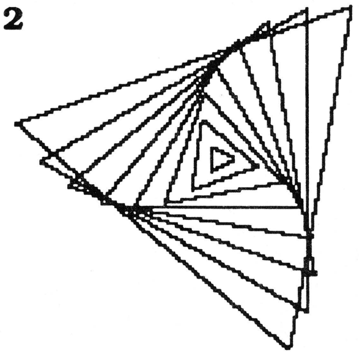
Using the X and Y rotation simultaneously can result in strange shapes; for example, if you use a rotation of 90 degrees for both the X and Y axis, you will get a single point, which is clearly not the right shape. If you set the X and Y rotations to the same value, you can make the whole polygon smaller, but it is easier to change the radius to get this effect. I reccommend you always use one or the other of the X and Y rotation, but noth both.

The Z axis can be thought of as a straight line emerging from the center of the polygon. Rotation around the Z axis means clockwise or counterclockwise rotation. Postive values of ZR rotate the polygon clockwise; negative values, counterclockwise. You can use Z rotation in conjunction with either X or Y rotation. The Z rotation is applied last, so you can rotate an elongated (or squashed) polygon around its center. Experiment! I am sure you will like the results.

How it Works

Trigonometry (the measurement of triangles) and circles are closely related. Given an angle, we can calculate, by using the SIN (sine) and COS (cosine) functions, the point on a unit circle (a circle with a radius of 1) associated with that angle. Scaling that point outward by multiplying by the radius allows us to

Figure 2



place the point on a circle of any radius. If we calculate six evenly spaced points on the circle, we can connect the six points to draw the inscribed hexagon. This is exactly how the routine works, except that things are further complicated by rotation.

To implement rotation around the X and Y axes, we calculate the cosine of the rotation angle and use it as a multiplier to "shorten" the appropriate axis. For example, if you pass the routine a rotation about the Y axis of 45 degrees, it calculates a cosine of about .707107, and multiplies each X coordinate by that amount, making the polygon about 1/3 narrower than it would usually be.

Rotation arount the Z axis is somewhat trickier. Given a point (X, Y), and an angle ZR, we can calculate a new point (X0, Y0) rotated about the Z axis with the following formulas:

```
X0 = X * COS (ZR) - Y * SIN (ZR)
Y0 = Y * COS (ZR) - X * SIN (ZR)
```

Since ZX, ZY, and ZR are constant for a given polygon, their sines and cosines are constant as well. Thus, in lines 1020-1050, we can pre-calculate the ratios needed to handle the rotation, before the actual plotting loop begins. (By the way, the expression inside the parentheses in these lines converts the value specified in angles to a value in radians. Most people like to think in degrees, but mathematicians and Applesoft like to think in radians.) Since SIN and COS are slow functions, our routine runs much faster than if we redundantly calculated the needed sines and cosines each time through the loop.

In line 1060 we calculate the step value for the loop by dividing the circle (which contains two times pi radians) into N even portions. The end point for the loop is defined to be just a wee bit past two pi radians, so that we can be sure that the final line segment back to the first point will be plotted.

Lines 1070-1150 compose the main plotting loop. Lines 1080 and 1090 determine the coordinates of the current point, adjusting for the radius and the X and Y axis rotation values. Lines 1100 and 1110 adjust for rotation about the Z axis. Line 1120 add in the center coordinates of the polygon (until this point, the polygon's points are calculated as if its center was at 0,0). Line 1130 handles the case of the first point being plotted, we can't use HPLOT TO until at least one point has been plotted, and this line takes care of that. Line 1140 plays connect-the-dots on our calculated points.

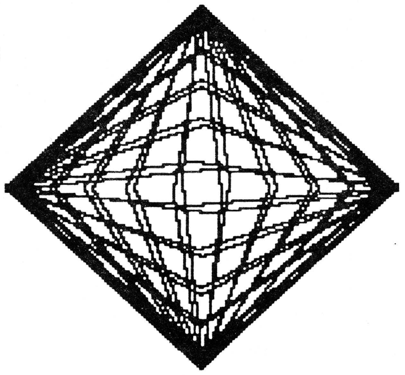
And that's all there is to it.

Exciting Graphics

You can use the polygon routine to draw many exciting pictures. Consider, for example, Listing 2. You must type in listing 1 first, then listing 2, because listing 2 requires the polygon routine. Listing 2 draws pictures like Figure 1 and Figure 2. This is done by choosing a random polygon from a triangle to an octagon, varying the radius from 5 to 75 in steps of 10, and by changing the Z rotation (by a randomly chosen increment) each time the size is increased.

Listing 3 draws pictures like Figure 3 and Figure 4. Once again, a random polygon is chosen, but the size and Z rotation stay the same; instead, we vary the X and Y rotations by a pre-chosen random increment.

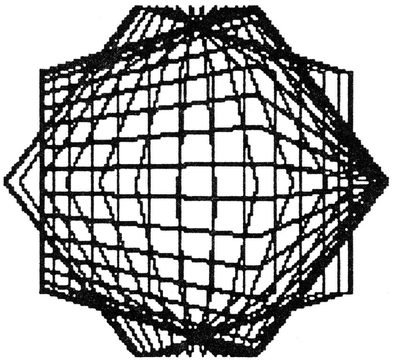
Figure 3



I am certain you can come up with many more such "computer art" programs yourself. Give it a try; it is easy! You might also try playing with the polygon routine itself to create variations on each artistic "theme". For example, you could switch EZ and EV in line 1110, or change line 1080 to read X = R * COS (I) * COS (I) * EY. Be sure to experiment with different values of all the rotation factors. You probably won't get regular polygons anymore if you change the polygon routine, but what you do get might be even more visually interesting! Anything goes as long as you are careful not to let the coordinates get out of range and generate an "ILLEGAL QUANTITY" error.

The routine is not fast enough for animation, but you could try anyway. You could also use an array to store the lines instead of plotting them, then "play back" the lines stored in the array at a higher speed.

Figure 4



Listing 1: Polygon Subroutine

```
1000  REM draw polygon
1010 P2 = 6.2831853: REM 2pi
1020 EX =  COS (RX / 360 * P2)
1030 EY =  COS (RY / 360 * P2)
1040 EZ =  COS (RZ / 360 * P2)
1050 EV =  SIN (RZ / 360 * P2)
1060 S = P2 / N:E = P2 + .01
1070  FOR I = 0 TO E STEP S
1080 X = R *  COS (I) * EY
1090 Y = R *  SIN (I) * EX
1100 X0 = X * EZ - Y * EV
1110 Y0 = X * EV + Y * EZ
1120 X = X0 + XC:Y = Y0 + YC
1130  IF I = 0 THEN  HPLOT X,Y
1140  HPLOT  TO X,Y
1150  NEXT
1160  RETURN
```

Listing 2: Art Program 1

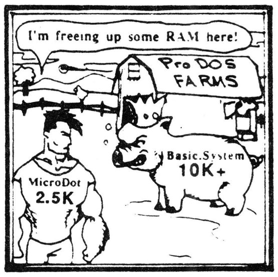
```
10 HGR2 : HCOLOR= 3
20 RX = 0:RY = 0:RZ = 0
30 XC = 139:YC = 95
40 N =  INT ( RND (1) * 6) + 3
45 D =  INT ( RND (1) * 10) + 5
50 FOR R = 5 TO 75 STEP 10
60 GOSUB 1000
70 RZ = RZ + D
80 NEXT
90 FOR I = 1 TO 3000: NEXT
100 RUN
```

Listing 3: Art Program 2

```
10 HGR2 : HCOLOR= 3
20 RX = 0:RY = 0:RZ = 0
30 XC = 139:YC = 95:R = 75
40 N =  INT ( RND (1) * 6) + 3
45 D =  INT ( RND (1) * 10) + 5
50 FOR RX = 0 TO 180 STEP D
60 GOSUB 1000
70 NEXT
80 RX = 0
90 FOR RY = 0 TO 180 STEP D
100 GOSUB 1000
110 NEXT
120 FOR I = 1 TO 3000: NEXT
130 RUN
```

MicroDot just $ 29.95 plus $2.50 S&H

The Logical Replacement for BASIC.SYSTEM



Just **2.5K** in size, but more powerful than BASIC.SYSTEM. Imagine doing BASIC overlays simply by specifying the file name and the line number where you want to overlay. How about loading an array of directory names at machine language speed. You get this and total control over ProDOS that is impossible with BASIC.SYSTEM. Works with Program Writer ($42.45. Both for $59.95 + S&H). Love it or get your money back! Inexpensive publishers' licenses.

Free Catalog and Details — Dealer Inquiries Invited

Kitchen Sink Software, Inc
903 Knebworth Ct. Dept. 8
Westerville, OH 43081
(614) 891-2111

TM

IIgs Graphics

Apple Preferred Pics in Pascal

Have it Your Way - & Their Way

by Phil Doto

Apple II GS Super Hi-Res pictures can be (and are) stored in a wide variety of formats. There are screen sized pictures and page size pictures; compressed pictures and uncompressed pictures; 320 mode pictures and 640 mode pictures; pictures that use only one palette and pictures that use all 16 palettes. To help bring order out of potential chaos, Apple recommends that all graphics programs support a file format known, oddly enough, as "Apple Preferred" format.

Apple Preferred Format (APF) pictures are stored on disk as filetype $C0, auxiliary type $0002. Unlike other file formats, APF files make very few assumptions about the picture. These files can contain any picture that QuickDraw II can handle and all the information needed to reconstruct the picture is stored in the file. These files can also store other graphics information, such as a library of palettes, and the format is very flexible and can easily be extended to contain additional information.

In this article, I'll be discussing how to read an APF file, interpret the information, and display the results. All of the code is in TML Pascal II and uses Apple's standard Pascal interfaces. I've tried to design the code in a way that is easy to follow and understand, even if you normally work in another language. My hope is to give you more than just some code that you can "cut and paste" into your programs. I want to leave you with an understanding of APF files that will allow you to adapt and modify these routines to work with your application.

Loading the File

I could read the information from the file a little bit at a time, processing it as I go; but, it's less confusing (at least to me) if I load the file first and then process it. Listing 1 is a Pascal function that will do the job of getting a file off the disk and into computer memory.

```
   {*────── LISTING 1 ──────*}

function LoadFile(theGSPath:GSString255;
                  VAR theHandle:Handle;
                  VAR theSize:longint):boolean;

var   paramsOpen  :   OpenRecGS;
      paramsRead  :   IORecGS;
      paramsClose :   RefNumRecGS;
      paramsEOF   :   EOFRecGS;
      i :             integer;

begin

      LoadFile := false;

   { open the file }
      paramsOpen.pcount := 2;
      paramsOpen.pathname := @theGSPath;
      OpenGS(paramsOpen);
      theError := _ToolErr;
      if theError <> noError then begin
           ReportError(theError);
           exit(LoadFile);
           end;

   { set up the close params }
      paramsClose.pcount := 1;
      paramsClose.refnum := paramsOpen.refnum;

   { find out how many bytes to read }
      paramsEOF.pcount := 2;
      paramsEOF.refnum := paramsOpen.refnum;
      GetEOFGS(paramsEOF);
      theError := _ToolErr;
      if theError <> noError then begin
           ReportError(theError);
           CloseGS(paramsClose);
           exit(LoadFile);
           end;

   { allocate memory for the file }
      theHandle := NewHandle(paramsEOF.eof,
                             myMemoryID,
                             attrLocked,
                             nil);
      theError := _ToolErr;
      if theError <> noError then begin
           ReportError(theError);
           CloseGS(paramsClose);
           exit(LoadFile);
           end;
```

```
  { read it }
     paramsRead.pcount := 4;
     paramsRead.refnum := paramsOpen.refnum;
     paramsRead.databuffer := theHandle^;
     paramsRead.requestCount := paramsEOF.eof;
     ReadGS(paramsRead);
     theError := _ToolErr;
     if theError <> noError then begin
          ReportError(theError);
          CloseGS(paramsClose);
          DisposeHandle(theHandle);
          exit(LoadFile);
          end;

     theSize := paramsEOF.eof;

     Loadfile := true;        { we got it loaded, }
     CloseGS(paramsClose);    { so close the file }

end;

   {*———————————————*}
```

This function requires three parameters: a GS/OS string for the pathname of the file I want to load, a variable handle parameter, and a variable longint parameter. If all goes well, the function will return true, the handle will be a handle to the memory block assigned to the file, and the longint parameter will contain the file length.

Of course, I could load a file using standard Pascal I/O statements, but it's usually a lot faster to make direct calls to the operating system. For each GS/OS call, I set up an appropriate parameter block and then make the call. You'll notice that I set up the close parameter block right after opening the file. That's so if I get an error later on, we can easily close the file on the way out.

Reading a file with GS/OS is really very straight forward. I open the file, get the EOF to find out how many bytes to read, call NewHandle to allocate memory for the file, read the file into the space the memory manager gave us, and then close the file.

Data in the APF files

OK, I can get the file into memory, but what data is stored in the APF file and how do I get at it? APF file data is stored in variable length blocks. The first thing in each block is a longint that specifies how long that block is (including the length longint itself). The length is followed by a string with the name of the block. This string is a ordinary Pascal string that starts with a length byte.

The picture, if there is one, is stored in a block called "MAIN". Apple has also defined blocks for patterns ("PATS"), palettes ("PALETTES"), and document drawing patterns ("SCIB"), and applications can define other blocks for their own uses. Notice that there doesn't have to be a MAIN block in a legal APF file, but it will be there if the file contains a picture. To load the picture, I need to find the MAIN block and extract the picture data from it.

In order to make it easy for us to manipulate the data in our Pascal program, I can extract the information from the file I've loaded and store the result in a Pascal data structure. First, I need to define a data type that will hold all the necessary information.

```
type PicRecord =
       record
          ImageHandle :        handle;
          MasterMode :         integer;
          PixelsPerScanLine :  integer;
          NumScanLines :       integer;
          LineSCB : array[0..MaxLine] of integer;
          NumPalettes :        integer;
          Palette : array[0..15] of ColorTable;
       end;
```

Let's take a look at each of the fields in this record.

ImageHandle is a handle to a block of memory that contains the actual pixel image. The pixel image is compressed in the MAIN block of the APF file, so our program will need to allocate memory for the uncompressed image and then unpack the image into this space. But, the image by itself isn't enough; I need the rest of the information in the PicRecord so I can display it properly.

MasterMode tells us what the global mode of the picture is. Normally, this will be either $00 (320 mode) or $80 (640 mode). I need to pass this value to the SetMasterSCB call before displaying the picture.

PixelsPerScanLine is the number of pixels to display on each scan line. Please note that this is the width of the picture and not the width of the screen. A picture in an APF file can be wider or narrower than the screen. A picture with 640 pixels per scan line could be a 640 mode picture, but it could also be a 320 mode picture that's two screens wide.

NumScanLines is the height of the picture. As with the width of the picture, any value is possible. The picture may be taller or shorter than the 200 scan lines that can be displayed on the screen.

Each scan line has a Scan line Control Byte (SCB) associated with it. This byte specifies things about how that scan line should be displayed. This includes the mode, interrupt status, fill mode status, and which color table to use. The array used to store the SCBs for the picture really should be dimensioned [0..NumScanLines - 1]. Unfortunately, I don't know what NumScanLines is going to be and Pascal doesn't support dynamic dimensioning of arrays. The best I can do is size the array large enough to handle anything we are apt to encounter by setting an appropriately large value for the constant MaxLine.

NumPalettes is the number of palettes that have been saved with the picture. The APF file type doesn't put any restriction on this number, but I can normally expect it to be between 1 and 16.

Finally, I have an array of color tables. As with the SCB array, this should be dimensioned [0..NumPalettes - 1]. However, since the current hardware only supports 16 palettes, it should be safe for us to use a [0..15] array.

Extracting the Data

Now that I know what I'm looking for, let's go get it. Listing 2 is a Pascal function that will load an APF file (using the function discussed earlier), extract the data for our PicRecord, and unpack the image.

```
   {*——————— LISTING 2 ———————*}

function LoadAPF(thePathName : GSString255;
                 VAR thePic:PicRecord): boolean;

     { This function loads the APF graphic  }
     { specified by the GSOS string.        }
     { If successful, true is returned and  }
     { all fields of the PicRecord are set. }

type longintPtr   =  ^longint;

var  myBufferHndl : Handle;
     i,j,
     PixelsPerByte,
     bytesDone,
     srcSize,
     dstSize   :    integer;
     srcBuffer,
     dstBuffer :    ptr;
     linesize,
     picsize,
     address,
     EndAddress,
     size,
     length,
     offset     :     longint;
     kind       :     str255;

begin

  LoadAPF := false;

  { load the file and find the buffer }

  if LoadFile(thePathName,
              myBufferHndl,
              size) then begin

    address := longint(myBufferHndl^);
    EndAddress := address + size;

   { find the MAIN block }

    repeat

       length := longintPtr(address)^;
       offset := 4;
       kind := StringPtr(address + offset)^;
       if kind <> "MAIN"
          then address := address + length;

    until (kind = "MAIN")
          or (address >= endAddress);

    if kind <> "MAIN" then begin
       DisposeHandle(myBufferHndl);
       exit(LoadAPF);
       end;

   { read the picture data }

    with thePic do begin

       offset := 9;
       MasterMode := intPtr(address + offset)^;

       if BAnd(MasterMode,$80) = 0
          then PixelsPerByte := 2    { 320 mode }
          else PixelsPerByte := 4;   { 640 mode }

       offset := 11;
       PixelsPerScanLine :=
          intPtr(address + offset)^;

       offset := 13;
       NumPalettes := intPtr(address + offset)^;

       offset := 15;
       if (NumPalettes > 0) then begin
          for i := 0 to NumPalettes-1 do begin
             Palette[i] :=
                ColorTablePtr(address + offset)^;
             offset := offset + 32;
          end;{for}
       end;(if)

      NumScanLines := intPtr(address + offset)^;
   { get memory for the pixel image }

        LineSize := PixelsPerScanLine
                         div PixelsPerByte;
        if LineSize mod 8 <> 0 then
           LineSize := LineSize
                       + 8
                       - (LineSize mod 8);

        PicSize := NumScanLines * LineSize;

        ImageHandle := NewHandle(PicSize,
                                 myMemoryID,
                                 attrLocked,
                                 nil);
        theError := _ToolErr;
        if theError <> noError then begin
           ReportError(theError);
           DisposeHandle(myBufferHndl);
           exit(LoadAPF);
           end;

   { unpack the picture }

        j := NumScanLines - 1;
        if j > MaxLine then j := Maxline;
        offset := offset + 2;

        srcBuffer := ptr(address
                         + offset
                         + (4 * NumScanLines));

        dstBuffer := ImageHandle^;

        for i := 0 to j do begin
           srcSize := intPtr(address + offset)^;
           LineSCB[i] :=
              intPtr(address + offset + 2)^;
           dstSize := LineSize;
           bytesDone := UnPackBytes(srcBuffer,
                                    srcSize,
                                    dstBuffer,
                                    dstSize);
           srcBuffer :=
             ptr(longint(srcBuffer) + bytesdone);
           offset := offset + 4;
        end; {for}
     end; {with}

     LoadAPF := true;
     DisposeHandle(myBufferHndl);

  end;

end;

    (*———————————————————*)
```

After the file is loaded, I extract the address from the handle and save it as a longint variable to use for pointer math. It is often said that pointer math is difficult in Pascal, but it isn't that bad if you save the address as a longint and then typecast it to the appropriate pointer type when you need to use it as a pointer.

The first thing I need to do is find the MAIN block. I check the name of the block and if it isn't "MAIN", I add the block length to the address and try again. I repeat this until we either find the MAIN block or reach the end of the file.

Notice the way that the string is assigned to the Pascal string variable 'kind'. This is an example of the way that data is accessed throughout this routine.

```
     kind := StringPtr(address + offset)^;
```

Remember that address is a longint that is equal to the address of the block in memory and that the name string (kind) comes right after a longint (4 byte) variable. To read the kind string, I need to look 4 bytes into the block. In other words, this string is located at an offset of 4 into the block. If I add the offset of the data to the address of the block, I get the address of the data. Then I can typecast this value into a pointer type that matches the type of data that will be found at that location (in this case, a StringPtr). To get the actual data, I dereference the pointer by adding a ^ and assign the result to our variable.

If I locate a MAIN block, I use this same approach to read the information in the block. First, at an offset of 9, I find the MasterMode. I find out which mode the picture is in by testing bit 7 of the MasterMode. This tells us how many pixels will be stored in each byte of the pixel image. I will need to know this later, so I stick the appropriate value in the PixelsPerByte variable. Next, I find the PixelsPerScanLine followed by NumPalettes. The palettes are stored right after NumPalettes. Each color table uses 32 bytes, so I increment offset by 32 for each palette in the file. NumScanLines will be found right after the palettes.

I'm just about ready to unpack the picture, so I had better allocate some memory for the image. The amount of memory that I need is the number of scan lines times the number of bytes in each scan line. I can get the number of bytes in a scan line by dividing PixelsPer-ScanLine by PixelsPerByte. Well, this almost works. It might not come out even and, also, some of the toolbox routines require that the number of bytes in a scan line be evenly divisible by 8 (LineSize mod 8 must equal zero), so I have to adjust the LineSize by rounding up to the next value that is divisible by 8.

Once I've allocated the memory, I'm ready to unpack the picture. There are two things left in our MAIN block: a scan line directory and the packed scan lines. The scan line directory has two entries for each scan line, the number of bytes to unpack and the scan line control byte.

I will use the same 'address + offset' approach that I have been using all along to read the scan line directory and set up another pointer (srcBuffer) to the packed scan lines. Since address + offset currently points to the start of the scan line directory and each directory entry is 4 bytes, address + offset + (4 * NumScanLines) will point to the start of the packed data.

I unpack the image by feeding each scan line to the toolbox UnPackBytes routine. This routine requires 4 parameters. SrcBuffer is the pointer to the packed scan line. SrcSize is the number of bytes to unpack and I can get this from the scan line directory. DstBuffer is a pointer to the memory I've allocated for the image. The toolbox automatically updates this pointer for us each time UnPackBytes is called, so it will always point at the spot where the next line goes. DstSize is the size of the destination space. I could set this to the full size of the unpacked image and the toolbox will update it for us on each call. The only problem is that it's an integer sized variable and it's at least theoretically possible for an APF picture to be larger than that. I can get around this by setting it to the line size for each line that I unpack. Each time I call UnPackBytes, I use the returned value to update our srcBuffer pointer.

Each time through the loop, I grab the SCB for the line and put it into our array. Since I don't want to try and store more data than our array will hold, I limit the number of lines that I unpack to the MaxLine constant that I used to dimension the array.

After I've finished unpacking the picture, I call Dispose-Handle to deallocate the buffer that I've been using for the APF file. I don't need the APF file image that I loaded from the disk any more, since I have the unpacked image and all the data is in our PicRecord.

Displaying the Picture

At this point, I have a picture image and all the information needed to display it properly somewhere in RAM, but since you probably actually want to look at the picture, I'm not done yet.

Listing 3 includes some procedures that demonstrate one way to draw pictures from a desktop program. ShowPic sets things up and then calls DrawPic to actually draw the picture to the screen. ClosePic will return us to the normal desktop display.

```
   (*------ LISTING 3 ------*)

procedure DrawPic(thePic : PicRecord;
                  srcLoc : LocInfo;
                  srcRect : rect);

var  i,j : integer;

begin

   {set up the SCBs for screen to correspond }
   { to the picture scan lines being displayed. }

     j := srcRect.v1;
     for i := 0 to 199 do begin
         SetSCB[i,thePic.lineSCB[j]);
         j := j + 1;
         end;

    { and then copy the display rect }
    { to the current port. }

     PPToPort(@srcLoc,srcRect,0,0,0);

end;

   (*----------------------*)

procedure ShowPic(myPic : PicRecord;
                  VAR PicLoc : LocInfo;
                  VAR DisplayRect : rect;
                  VAR PicPort : grafPort);

var  i,N,myMode : integer;

begin

    { set up a LocInfo record }

    with PicLoc do begin
       portSCB := myPic.MasterMode;
       PtrToPixImage := myPic.ImageHandle^;
       if BAnd(myPic.MasterMode,$80) = 0
          then N := 2 else N := 4;
       width := myPic.PixelsPerScanLine div N;
       if width mod 8 <> 0
          then width := width + 8 - (width mod
8);
       boundsRect.h1 := 0;
       boundsRect.v1 := 0;
       boundsRect.h2 := myPic.PixelsPerScanLine;
       boundsRect.v2 := myPic.NumScanLines;
       end;

   { clear the desk }

    HideMenuBar;
   HideCursor;

  (get a port to display the picture. )
  ( first set the master SCB )
  ( and then open a port in the new mode. )

   SetMasterSCB(myPic.MasterMode);
   OpenPort(@PicPort);

  ( adjust the color tables for the picture )

   if myPic.NumPalettes > 0 then
      for i := 0 to myPic.NumPalettes - 1 do
         SetColorTable(i,myPic.Palette[i])
   else begin
      initColorTable(myPic.Palette[0]);
      SetColorTable(0,myPic.Palette[0]);
   end;

  ( set up a rectangle to display as much )
  ( of the picture as possible. )

   if N = 2 then myMode := 320 else myMode :=
640;
   with DisplayRect do begin
      h1 := 0;
      v1 := 0;
      if PicLoc.boundsRect.h2 > myMode
         then h2 := myMode
         else h2 := PicLoc.boundsRect.h2;
      if PicLoc.boundsRect.v2 > 200
         then v2 := 200
         else v2 := PicLoc.boundsRect.v2;
   end;

  ( and then show the picture )

   DrawPic(myPic,PicLoc,DisplayRect);
   PicShowing := true;

end;

    (*————————————————————*)

procedure ClosePic(VAR PicPort : grafPort);

var  StdColors : ColorTable;

begin

( close the display port )

     SetPort(WindOnePtr);
     ClosePort(@PicPort);

( set up for 640 mode )

     SetMasterSCB($80);
     SetAllSCBs($80);

( restore the standard color table )

     initColorTable(StdColors);
     SetColorTable(0,StdColors);

( show my stuff and redraw the desktop )

     ShowMenuBar;
     ShowCursor;
     ShowWindow(WindOnePtr);
     SelectWindow(WindOnePtr);
     RefreshDesktop(nil);
     PicShowing := false;

end;

    (*————————————————————*)
```

ShowPic uses the information in our PicRecord to set up a LocInfo record that I can use with QuickDraw II. Once again, I have to be careful that the width in bytes is divisible by 8; otherwise, this should be fairly straight forward.

Since I'm going to use the full screen for the picture, I can avoid some of the complications involved with a full mode change. First, I clear off the desktop. I close any open windows before calling the procedure, so I only need to get rid of the menu bar and the cursor. Next, I feed our picture's MasterMode to SetMasterSCB and then call OpenPort to get a full screen sized GrafPort that matches the picture's mode. OpenPort sets things up based on the MasterSCB, so I'll either get a 320 or 640 port depending on the MasterMode in our PicRecord.

I need to set the palettes to those in our PicRecord before I draw the picture. The SetColorTable call in a simple for loop will do the job. In the exceedingly rare, but legal, event of an APF file with zero palettes, I'll use a standard color table.

Before I can use PPToPort to copy the picture into our new port I need to define the source rectangle. I want to display as much of the picture as possible, so the rectangle's dimensions will either match the screen's dimensions or picture's dimensions, whichever are smaller. I pass this rectangle to DrawPic which draws the picture on the screen. Finally, I set a global flag so that other routines in the program to tell that a picture is being displayed.

DrawPic sets the SCBs for the screen to correspond to the SCBs for the lines of the picture and then calls PPToPort to copy the image to the screen. I've separated this routine from the ShowPic procedure to make it easy to scroll the picture. To see other parts of the picture, all I need to do is offset the display rectangle and call DrawPic again.

ClosePic starts out by resetting the current port (I've used a global window pointer in this example) and then closing the picture's port. I need to set some other port first because closing the current port is a no-no. Next, I set the MasterSCB and all line SCBs back to the program's mode. Then I can restore the default color table, make all our desktop stuff visible again, and redraw the desktop.

Modifications and Improvements

There you have it, Apple Preferred Format pictures that will support just about anything that the Apple II GS can display. I went through a lot of contortions to get the picture on the screen, but I wanted to keep these routines as general as possible and performance is still very acceptable. On my system (a Transwarped GS with a 60 mb SCSI hard drive), these routines will load, interpret, unpack, and display a screen sized picture in just over one second.

Still, there are many opportunities to modify and improve on this code. For one thing, I designed the PicRecord to help explain the contents the APF file. This is probably not the most efficient way to organize and store the data in your program. For example, you might want to set up a LocInfo record and store that instead of some of the more basic data. Also, there are many places that things can be simplified if you only need to deal with screen sized images or can put other restrictions on the pictures.

Insecticide

• *Them's The BRKs*, our article on 8-bit relocation, contained two errors. If you actually tried to assemble the code, you noticed that Merlin threw up its hands when it encountered the label ":proc" in lines 82, 87, 112, and 116. Replace ":proc" with "RTR_PROC" and the problem will go away. There were also errors in the RTR_DINS routine which caused it to hang; here is a corrected version:

```
RTR_DINS    php
            pha
            brk
            lda   RTR_OLDB
            pha
            brk
            lda   RTR_OLDB+1
            sta   $3F1
            pla
            sta   $3F0
            pla
            plp
            rts
```

The two corrections above were included on last month's *8/16* disk. Dave Lyons at Apple also pointed out that interrupts can wreak havoc with the RTR routine, particularly the code which figures out its own runtime address. If an interrupt occurs after the RTS at $FF58 is executed, but before the code which adjusts the stack pointer back downward is executed, the stack data the routine uses to calculate its runtime address will be corrupted. To avoid this, insert a SEI after line 19. To be safe, we should probably disable interrupts in the actual BRK handling code itself, as well; inserting a SEI after line 64 should do the trick. (In neither situation do we need to re-enable interrupts, since the existing code already takes care of saving and restoring the processor status register, thus doing that for us.)

• In David Guager's *Hardware Hacker* column of last month, our rendition of David's diagrams didn't turn out quite right. In Figure 1, the ground wire should go into the game port immediately above the number 3. The 5v line should line up with the number 2. Furthermore, in line 320 of the Biofeedback program listing, there should be a colon between the HTAB 11 and the PRINT statement.

The ZBasic Zealot

More MLI Madness & Working with Words

by Ross W. Lambert, Editor

I've spent a goodly amount of time over the last few weeks doing two things: 1) helping some folks deal with the ProDOS Machine Language Interface from ZBasic, and 2) playing with and adding to Chet Day's *Shem the Penman's Guide to Interactive Fiction.* Both duties have inspired me to explain a few things about each this month.

ZBasic2MLI

First, the ProDOS MLI to ZBasic connection... a grasp of the big picture will really help.

Here's a simple rule for you: *everybody* doing file I/O under ProDOS 8 has to access the MLI. Everybody. This means assembly language programs, ZBasic programs, Aztec C programs, Micol Advanced Basic programs, and even Applesoft programs via BASIC.SYSTEM. The syntax and available commands differ in each environment only because ZBasic and the gang provide a sort of custom interface between you and the MLI. But rest assured that, at the machine code level, they are each doing the same sorts of things whenever they are talking to ProDOS 8. This same principle applies to 16 bit software, too, as Mike Westerfield indirectly pointed out in his article this month. At the machine code level, a GS/OS call is a GS/OS call is a GS/OS call...

By providing us with the location of ZBasic's built-in parameter blocks and subroutines for MLI calls, language wiz Greg Branche (now with Apple, Inc.) gave us the ability to stick our toes into running water. That is, we can set up ZBasic's internal routines so that it calls the MLI for us, and in exactly the same manner that a standard ZBasic file I/O statement does. He also saved us a few bytes in that we don't need to reserve space elsewhere for a place to put the information passed to and from the MLI (such a data stash is called a parameter block).

Since the MLI is being accessed via ZBasic internal routines, ZBasic itself does not know or care who is making the call. This is important in its implications. For example, if I try to OPEN a file via the MLI (instead of the standard ZBasic OPEN) and the MLI returns an error, ZBasic will do what it always does. That is, it will fill in the ERROR variable with the error number. If I want to, I can let ZBasic handle the error. On page D-19 of the ProDOS appendix in the manual, Greg provided us with machine code that turns over error handling to ZBasic after an MLI call. You can mess with that if you want, but I find it much more straightforward to handle errors myself. It's just mo' betta all the way around, I think, especially if ERROR will tell me what has happened (which it does).

Skip This

If you're already an assembly language junkie, skip this section. From the drift of my phone calls of late, there are enough questions about accessing the MLI that a quick once over here might help a few of you.

And even if assembly language makes you nauseous, read on. There's surprisingly little assembly involved. In fact, you don't even need an assembler to call the MLI with ZBasic!

Because we're using some machine code living inside of ZBasic to access the MLI, we don't need to do as much as most folks do when making an MLI call. In a nutshell, all we have to do is POKE the number of parameters required for the call into $1F00, and then POKE the parameters themselves into the appropriate bytes thereafter (i.e. $1F01, $1F02, etc.)

A common source of boo boos at this stage is losing track of what goes where. To get the MLI to OPEN a file for us, for example, the manuals and books (*The ProDOS 8 Technical Reference Manual* and *Exploring GS/OS and ProDOS 8*) say that there are three parameters total. Two we give (pass) to ProDOS and one it gives back to us. Thus we must:

```
POKE &1F00,3 :REM POKE # of parms at $1F00
```

The first parm we must pass is a pointer to the file name. That's easy with VARPTR. Like so:

```
POKE WORD &1F01, VARPTR (FILE$)
```

The second parm is the address of the 1K I/O buffer ProDOS needs for a workspace. For this we could hunt around for 1K within our program or data space, but Greg was kind enough to tell us where ZBasic puts its own I/O buffers so that we could share them. As long as you don't overwrite the buffer of an open file, there is no problem with using ZBasic's buffers. You can figure out where a buffer is by multiplying 1K (1024) times the number of open files and subtracting that from $AC00. It's easier than it sounds...

```
BUFFER% = &AC00 - (FNUM * &400)
POKE WORD &1F03, BUFFER%
```

Note that, on 8 bit Apples, memory addresses are a word (16 bits) long, so I used POKE WORD instead of POKE. Also note that you have to watch out for how you've configured ZBasic. If you have told the compiler you're only going to have one file open at a time, you can only have one file open at time. If you put a buffer 1K below the first buffer you'll be trashing your own variables.

This is unpleasantness.

To actually make the call to the MLI, you need a MACHLG statement that looks like this:

```
MACHLG &A9,MLICallNum, &20, &0865
```

In our case, the MLI call number for the OPEN command is $C8. Put that in for MLICallNum and by George, you've got it. That's all there is to it.

Put that assembler away!

To review, then, there are three basic steps involved in putting the MLI to work for you in ZBasic:

• 1: POKE the number of parameters for the call you want to execute at $1F00.

• 2: POKE the rest of the parameters for any given call in the appropriate spots from $1F01 on.

• 3: Insert a `MACHLG &A9,MLICallNum, &20, &0865` right into your code.

Take a gander at Listing 1. The code there purposely sets up an error condition with an illegal file name. By running the beast you can see that, just as I promised, ZBasic 4.21 fills in ERROR and tells you that something is rotten in Denmark.

Immediately after that, the program takes a real file that is online (whose name you have to insert into the source first!) and then gets the file length using two different methods. In the first situation I accessed the MLI directly in the same manner I just described. In the second, I told ZBasic to go OPEN the file, give it a record length of one byte, and then tell me how many records there are. This effectively returns the file length. Because ZBasic I/O routines are doing some calculations while this is going on (so that we can use fixed length records), the direct-connect method via the MLI is significantly faster. Try building a loop that does 100 iterations of each method and then time them if you don't believe me.

Now all you've got to do is go out and buy Gary Little's *Exploring GS/OS and ProDOS 8* so you can make all sorts of off-the-wall MLI calls. For fun, try changing the filetypes and aux filetypes of all the files on your hard drive. Better yet, do the same thing on original copies of AppleWorks.

Hey man, this is the April Fool's edition, remember?

Shem on You

As I mentioned earlier, I've been writing some adventure games lately and have been looking for ways to add intelligence to the natural language interpretation abilities of my programs. The code in Listings 2 and 3 are the fruits of my labors (and research - they're adaptations from the *Microsoft QuickBasic Toolbox*).

An aside - many of us in Appledom forget that the (shudder) MS-DOS world is absolutely gargantuan in terms of sheer numbers of users and programmers. There is some good stuff going on over there in the software arena which we can benefit from if we put aside our biases for a few minutes. Just be sure and pick 'em up again! The hardware is really yucky.

Back at the ranch, natural language processing is an incredibly detailed subject and the subject of more than one doctoral thesis. In its fullest and purest form it is way over my head.

But I know what I want my software to do: it should be able to discern the meaning of as many common English phrases and directives as possible. Even with today's high speed CPUs, going "brute force" through every possible combination of letters is ridiculously slow. Therefore my program must be able to do some word parsing and intelligent guessing.

Here's how I did it (with apologies to all you computer science majors)...

For my purposes (and most others), the ability to pick out the individual words from a string typed at the keyboard is the first step towards figuring out the meaning behind the command.

In an (English) adventure game setting, the first word typed is usually the verb - "Drop the phasor", for example. The second word is often an article ("a", "an", or "the"), and the third word is the direct object - the object or person we should do something to. With some decent word parsing code we can yank out each word from the command line and compare it against a list of logical responses, acting accordingly if a match is found.

By the way, recent advances in adventure game development have greatly improved the command line ("you type it") interface, providing menus and mouse support for quick direction changes, etc. Nevertheless, I still think the best pieces of software let me "talk" to the computer also, making creative decisions on the fly via the keyboard. It's a tough job, and not many programs do it well.

FN ParseWord (Listing 2) can be a decent first step on your parsing path. The function makes good use of ZBasic's INSTR statement, scanning the target string for the first instance of a "seperator" character. In most instances, the seperator is a space. For added flexibility, however, you can define as many seperators as you want. Thus spaces, hyphens, backslashes, and just about anything else can be used to define word breaks.

Because all variables are global in ZBasic, it is useful in this function to "officially" return one string - the first word found - in TheWord$, and also the rest of the string - i.e. everything to the right of the first word - in Source$. In this manner, repeated calls to the function can rip apart any string into its component words.

Listing 3 is a beast of a different color. There are times, when dealing with human language, that you need to evaluate a set of strings and find out which one is closest to something your program can understand.

FN BestMatchStr scans the INDEX$ array and compares the strings within it to a Target$. For added flexibility, you can pass the function the array element to start with, the number of comparisons to make, and whether or not to check for case. If the parm, CaseSensitive is boolean true (i.e. equal to -1), then the function will not count "roscoe" to be as close a match to "Ross" as "Roscoe".

The actual string comparison code is rather interesting, I think. The routine works by creating a score for each element in the comparison array. The score is based on both the number of character matches and the length of each match.

In such a "length weighted" scheme, if the target string was "CAT", "CAR" would have a higher score than "KTA" even though they each matched on two characters. This is because CAR matched with two *consecutive* characters.

A Caveat

Even though FN BestMatchStr will tell you which string is the closest to a target string, it will not give any indication if the closest match is "close enough". You would probably have to determine a "minimum matching score", a process that is best left to your individual applications.

Besides, I am out of space and time (oooh, broke in four dimensions!). Until next time, then, remember:

1) Never tell a telephone operator that you'd like to CALL -958,

2) Never POKE anything you ought not to in a parameter block (ouch!),

and

3) Never underestimate the power of BASIC.

Listing 1 : ZBasic ProDOS 8 MLI Calls

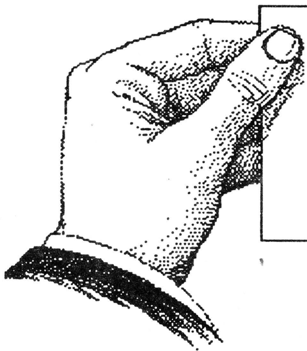
```
 REM
 REM
 REM   ZBasic ProDOS 8 MLI Stuff
 REM   by Ross W. Lambert, Editor
 REM   This baby¬s public domain
 REM
 REM
 :
 :
 DIM 65 FILE$
 :
 REM
 REM     Define Long Functions
 REM
 REM
 :
 LONG FN OPEN_FILE (FILE$,FNUM)
   BUFFER% = &AC00 - (FNUM * &400) :REM get 1K
            I/O buffer for ProDOS from Z
   POKE &1F00,3                     :REM three
             parms for this call
   POKE WORD &1F01, VARPTR (FILE$) :REM pass
             pointer to filename
   POKE WORD &1F03, BUFFER%         :REM tell
             ProDOS where buffer is
   MACHLG &A9, &C8, &20, &0865      :REM Open
             the file
   REF_NUM = PEEK (&1F05)           :REM Get
             ProDOS reference number
 END FN = REF_NUM
 :
 :
 LONG FN GET_EOF! (REF_NUM)
   POKE &1F00,2     :REM two parms for this call
   POKE &1F01, REF_NUM
   MACHLG &A9, &D1, &20, &0865:REM make the call
   FILE_LEN! = PEEK WORD (&1F02) + PEEK (&1F04)
            * 65536.0 : REM length of file
 END FN = FILE_LEN!
 :
 :
 REM
 REM          Main Program
 REM
 :
 :
 REM Intentionally create an error to see what
            ZBasic does
 :
 FILE$ = "12345"
 RefNum = FN OPEN_FILE(FILE$,1)
 IF ERROR <> 0 THEN PRINT "You boogered that
up!" : INPUT R$
 ERROR = 0
 :
 FILE$ = "Your File Here"  : REM 44 Put the name
of an available file here 44
 :
 RefNum = FN OPEN_FILE(FILE$,1)
 IF ERROR <> 0 THEN GOTO "Fatal Error"
 :
 FileLen! = FN GET_EOF!(RefNum)
 CLOSE
 :
 PRINT "The length of ";FILE$;" is: ";FileLen!;"
          bytes."
 :
 REM If we open with a 1 byte record length, we
          can use LOF to calculate
 REM the number of records (which is the number
          of bytes in this case)
 :
 OPEN "I",1,FILE$,1
 :
 PRINT "According to ZBasic, the file length is:
"; LOF(1)
 CLOSE
 :
 END
 :
 "Fatal Error"
 PRINT ERRMSG$(ERROR)
 STOP
:
:
```

Listing 2: FN ParseWord

```
 REM
 REM
 REM   FN ParseWord Example
 REM   by Ross W. Lambert, Editor
 REM   Copyright (C) 1989-90
 REM   Most Rights Reserved
 REM
 REM
 :
 DIM 2 Char$,OldChar$,20 Sep$,80 TheWord$ :REM
          lengths are pretty arbitrary
 :
 :
 LONG FN ParseWord$ (Source$,Sep$)
  TheWord$ = ""
  SepLen = LEN (Sep$):IF SepLen=0 THEN "ExitFN"
  SubLen = LEN (Source$)    : REM grab length of
            subject string
  IF SubLen = 0 THEN "ExitFN"
  FOR Char = 1 TO SubLen : REM loop through each
            character
     Char$ = MID$(Source$,Char,1)
 : REM we got us a word break
     LONG IF INSTR(1,Sep$,Char$) AND Char$ <>
            OldChar$
       IF Char = 1 THEN "NextChar": REM ignore a
            leading space
       TheWord$ = LEFT$(Source$,Char-1)
       Source$ = RIGHT$(Source$,SubLen-Char)
       GOTO "ExitFN"
     END IF
:
"NextChar"
     OldChar$ = Char$
  NEXT        :REM get next char in source string
"ExitFN"
  IF TheWord$ = "" THEN TheWord$=Source$:Source$
          = ""
END FN = TheWord$ :REM note Source$ holds rest
          of str
 REM ▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬
 REM          Main Program
 REM ▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬
 :
 TheStr$ = "Hit the troll with the rock.":REM
           typical adventure game string
 Sep$ = CHR$(32):REM space only separator here
 Source$ = TheStr$ :REM save a copy of the
           original str!
 :
REM loop through and print the first word on the
REM left and the remaining string on the right
REM until we've parsed every word.
 :
DO
  Count = Count + 1
  TheWord$ = FN ParseWord$ (Source$,Sep$)
  PRINT TheWord$,Source$
UNTIL LEN (Source$) = 0
:
INPUT R$
END
```

Listing 3: FN BestMatchStr

```
REM ▬▬▬▬
REM
REM   FN BestMatchStr Example
REM   by Ross W. Lambert, Editor
REM   Copyright (C) 1989-90
REM   Most Rights Reserved
REM
REM ▬▬▬▬
 :
 :
REM NOTE: Place the subject strings in the
REM INDEX$(X) array before calling the function.
REM
REM VARIABLES: Target$ - str others strive to be
REM            WannaBe$ - the subject strings
REM            NumComps - # of strings in INDEX$
REM            CaseSensitive  - if True, caps
           different than lower case
REM            StrScore1 - HIGHEST score for
           subject strings SO FAR
REM            StrScore2 - Pointer to HIGHEST
           RATED STRING.
REM            StrScoreX - Running total of
           current comparison.
REM            TargetLen - length of target str
 :
REM ========================================
 :
 :
LONG FNBestMatchStr(Target$,NumComps,CaseSensitive
 TargetLen = LEN(Target$)
 StrScore1 = 0   :REM You need this if you call
           multiple times
 FOR X = 0 TO NumComps-1
   LONG IF NOT CaseSensitive
     WannaBe$ = UCASE$(INDEX$(X))
   XELSE
     WannaBe$ = INDEX$(X)
   END IF
   StrScoreX = 0
   FOR I = 1 TO TargetLen
     FOR J = 1 TO TargetLen - I + 1
       Temp$ = MID$(Target$,J,I)
       LONG IF INSTR (1,WannaBe$,Temp$)
          StrScoreX = StrScoreX + (2*I)
       END IF
     NEXT
   NEXT
   IF StrScoreX > StrScore1 THEN StrScore1 =
           StrScoreX : StrScore2 = X
 NEXT

 END FN = StrScore2
REM Returns # of string with highest score
 :
REM ▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬
REM          Main Program
REM ▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬
 :
CLEAR 2000 :REM For INDEX$ array

INDEX$(0) = "This is a real test."
INDEX$(1) = "This is the first test."
INDEX$(2) = "This is a very real test."
INDEX$(3) = "This is the third test."
INDEX$(4) = "is is is is is is is is is is is
is" : REM check for weirdness
Target$ = "This is my real test."
 :
PRINT "The candidate strings:"
PRINT
FOR X = 0 TO 4
  PRINT INDEX$(X)
NEXT
PRINT
PRINT "The target:"
PRINT Target$
 :
BestStr = FN BestMatchStr (Target$,5,-1) :REM
           compare 5 strings, check case
 :
PRINT
PRINT "And the winner is string ";BestStr
PRINT INDEX$(BestStr)
 :
INPUT R$ : REM pause until CR
END
```



Whenever you place an order with an Apple II hardware or software company, tell them you saw it in 8/16 (whether they're advertisers or not - hehehe).

OrcaStrations

No Fits With Inits: Writing an Init From High Level Languages

By Mike Westerfield, The ByteWorks

Editor: Mike undoubtedly needs no introduction, but in my typically redundant manner I'll do it anyway: Mike is the author of numerous popular programming environments and languages for both 8 bit and 16 bit Apple II's, including the Orca assemblers, Orca C, and Orca Pascal. His company, The ByteWorks, has also produced some interesting lesser known packages like BytePaint (a DHR paint and shape drawing program), Voyager, and a neat GS product I saw at the last AppleFest called "The Talking Storybook". Today's trivia: The ByteWorks started naming their products "Orca" -whatever because their first assembler, Orca/M, was really "macro" spelled backwards. And you thought Mike was into whales!

Don't Quit! Return!

The first kind of program most people learn to write is either a stand-alone program that can be launched from the Finder, or a shell program that runs from the APW or ORCA shell. In either case, at the end of the program, you use a Quit call to return to the Finder or shell. You can also use an RTL to return to the shell (although not the Finder). To keep things simple, though, compilers exit using a Quit call, since that works with either type of program.

There have always been programs that had to finish with an RTL, though. The most common example of these are desk accessories, both classic desk accessories and new desk accessories. Compilers generally have some sort of directive to help you create the special code and headers that must be used with CDAs and NDAs. There are several cases, though, that the compilers do not handle. GS/OS lets you write programs called startup files, or inits. These are called as GS/OS boots, when you turn on your computer. For the most part, an init works just like any other program. The main difference is that you use an RTL to get back to GS/OS, instead of a Quit call. CDevs, the modules called by the new control panel, have the same requirement, as do the programs called by HyperStudio.

At first, most of the folks delving into these corners of the operating system were using assembly language, where you can return any way you want, but recently we have had more and more questions from people trying to write these kinds of programs from Pascal and C. This article explores how this is done. First, we will look at how to return with an RTL from Pascal and C. With the basics out of the way, we will write a short init in C.

RTL from C

When you compile a C program from ORCA/C, the compiler creates two object files, called a root file and a dot-a file. These files are sent to the linker, which turns them into an executable program. The reason that there are two files is tied up in the way partial compilation works, but it turns out that it is very handy for our purposes. The root file has the preamble code that initializes the run-time environment for the compiled functions, while the dot-a file has all of the functions you wrote in C. A disassembly of the root file looks like this:

Listing 1: Standard ORCA/C Root File

```
         keep   ccroot
         mcopy  ccroot.macros
         case   on
~_ROOT   start

         ph2    #$2000    ask for 8K of stack space
         jsl    ~_BWSTARTUP  set compilr environ.
         ph2    #~GLOBALS|-16  set data bank reg.
         plb
         plb
         jsl    ~C_STARTUP  executePascal program
         jsl    main
         jsl    ~C_SHUTDOWN
         end
```

It is ~_BWSTARTUP that does most of the work to set up the environment. It allocates a stack for local variables that is 8096 bytes long ($2000), starts up the memory manager that is used by Pascal and C, and does a few other housekeeping chores. You change the size of the run-time stack by changing the value pushed at the beginning of the subroutine. It then sets up the data bank register, and calls ~C_STARTUP. ~C_STARTUP is peculiar to the C language. This is where the command line is read and parsed for later use by argc and argv, and where C-specific initialization is done. The jsl to main calls your main function, which is the entry point to every C program. Finally, a call is made to ~C_SHUTDOWN. The last thing ~C_SHUTDOWN does is jump to ~QUIT, the exit point for all high-level languages. It is ~QUIT that we need to change.

~QUIT is located in the run-time library, embedded at the end of a segment that contains a number of global variables used by the compiler and its run-time library. Like all library subroutines, we can replace the one in the standard library with one of our own by just using the same name. The linker will use our substitute routine in place of the library routine. The replacement routine will use an rtl instead of a quit call, but this does present one problem. You can quit from anywhere in a program; GS/OS repairs the stack for itself. To do an rtl, you have to make sure that the stack register is exactly what it was when the program was called. The easiest way to make sure this happens is to save the stack register at the beginning of our root segment. To do that, we will replace the root file created by the C compiler with one of our own. Here's the replacement:

Listing 2: Modified ORCA/C Root File

```
         keep   ccroot
         mcopy  ccroot.macros
         case   on
~_ROOT   start

         tsc              save entry stack value
         sta    >~QUITSTACK
         ph2    #$2000    ask for 8K stack space
         jsl    ~_BWSTARTUP set compiler environm.
         ph2    #~GLOBALS|-16 set data bank reg
         plb
         plb
         jsl    ~C_STARTUP    execute Pascal prog
         jsl    main
         jsl    ~C_SHUTDOWN
         end
```

The only other step is to replace ~QUIT with code that restores the stack register to the value saved in ~QUITSTACK and returns with an RTL. The normal shut-down process used by the compiler will do all of the other clean-up, like disposing of our stack space, deallocating any memory, and so forth. The complete replacement subroutine, along with the global variables that appear in the same module, is show below.

Listing 3: Modified Library Subroutine With RTL

```
         mcopy common.macros
****************************************************************
*
*  ~_BWCommon - Global data for the compiler
*
****************************************************************
*
~_BWCommon start
;
;  Misc. variables
;
~CommandLine entry   ;addr of the shell cmd line
         ds    4
~EOFInput entry      ;end of file flag for input
         ds    2
~EOLNInput entry     ;end of line flag for input
         ds    2
ErrorOutput entry    ;error output file variable
         dc    a4'~ErrorOutputChar'
~ErrorOutputChar entry ;error output file buffer
         ds    2
Input    entry      ;standard input file variable
         dc    a4'~InputChar'
~InputChar entry    ;standard input file buffer
         ds    2
~MinStack entry     ;lowest resrved bnk zero addr
         ds    2
Output   entry      ;standard output file variable
         dc    a4'~OutputChar'
~OutputChar entry ;standard output file buffer
         ds    2
~RealVal entry ;last real value returned by a fn
         ds    10
~ThisFile entry ;ptr to current file variable
         ds    4
~ToolError entry      ;last error in a tool call
         ds    2
~User_ID entry        ;user ID
         ds    2
ioFlag   entry        ;input output flag
         ds    2
~StringList entry     ;string buffer list
         ds    4
;
;  Traceback variables
;
~ProcList entry       ;traceback list head
         ds    4
~LineNumber entry     ;current line number
         ds    2
~ProcName entry       ;current procedure name
         ds    12
```

```
;
;  Universal quit code
;
~Quit     entry
          pha                 ;save the return code
          jsl    ~MM_Init     ;zero the memory mgr
          ph2    >~User_ID    ;dispose of any re-
maining memory
          _DisposeAll         ;allocated by mem mgr
          plx                 ;restore return code
          lda    >~QuitStack;rest stack register
          tcs
          txa
          rtl                ;return to prog launcher

~QuitStack entry             ;S reg for quit
          ds     2
          end
```

Putting all of this mess together into a program is easier than it looks. The new version of ~_BWCOMMON can be appended right to the end of the C program with a #append; the system is smart enough to figure out that you changed languages, calling the compiler and assembler when appropriate. To try this out, we'll write the famous hello, world program so it does an rtl instead of a quit. With the #append at the end, the program looks like this:

Listing 4: Hello World from C

```
#pragma keep "test"

#include <types.h>
#include <misctool.h>
#include <stdio.h>

void main(void)

{
printf("Hello, world.\n");
}

#append "test.asm"
```

The assembly language file with ~_BWCommon should be called test.asm; if you change the name then you will, of course, have to change the name on the #append directive as well. The program is compiled with the compile command.

Next, we need to replace the root file created by the C compiler with our own version. To do that, assemble the replacement root file. I named the file ccroot.asm; you might want to do the same, to make it a bit easier to follow the article. Assembling this file produces an object file called ccroot.root. To replace the default root file, delete test.root (this is the name of the file created by the C compiler), and rename ccroot.root to be test.root. Finally, we link test.

This is, to put it mildly, a real mess to go through every time you compile the program. To avoid the hassle, the best thing to do is to encapsulate all of the commands in a script file. Here's the one I used. Type in it just like a program, then save it and set the language type to EXEC. To create and run your program, just type the name of the script file from the shell. (From PRIZM, you can do this from any window. If you are in the shell window, type the name of the script and press the return key. From some other window, you use the enter key, instead of the return key.)

Listing 5: Automating the Compile for C

```
compile test.cc
assemble ccroot.asm
delete test.root
rename ccroot.root test.root
link test keep=test
test
```

Once Again, from Pascal

ORCA/Pascal uses the same run-time environment as ORCA/C. Just like in C, the quit code is in ~_BWCommon. The only difference is that Pascal doesn't have to call language-specific routines to handle things like argc and argv. Basically, then, the only difference between creating a Pascal program that returns to the launcher with an RTL and doing the same thing for C is the code you put in the custom root file.

Here's the standard root file for a Pascal program. Right below it is the modified version that stores the value of the stack register for later use by the quit code.

Listing 6: Standard ORCA/Pascal Root File

```
          keep  pasroot
          mcopy pasroot.macros
```

```
~_Root   start

         ph2   #$2000 ;ask for 8K of stack space
         jsl   ~_BWStartUp ;set up complr envir.
         ph2   #~Globals|-16 ;set data bank reg
         plb
         plb
         jsl   ~_PasMain ;execute Pascal program
         lda   #0         ;return with no error
         jml   ~Quit
         end
```

Listing 7: Modified ORCA/Pascal Root File

```
         keep  pasroot
         mcopy pasroot.macros
~_Root   start

         tsc        ;save the entry stack value
         sta   >~QuitStack
         ph2   #$2000 ;ask for 8K of stack space
         jsl   ~_BWStartUp ;set complr envirment
         ph2   #~Globals|-16 ;set data bank reg
         plb
         plb
         jsl   ~_PasMain    ;execute Pascal prog
         lda   #0          ;return with no error
         jml   ~Quit
         end
```

A casual glance might make you try replacing the JML to ~Quit with an RTL, instead, but that is a bad idea. If you take a closer look at ~BWCommon, you will see that the quit code also shuts down the memory manager, and disposes of memory allocated by Pascal. Even if you put this code in your root file, too, there is a problem. Error exits from library subroutines are still going to leave the program by jumping to ~Quit. In other words, stick with the method outlined. It will save you a world of trouble.

Like C, Pascal has a compiler directive which can append an assembly language file. A simple hello, world program in Pascal, with the append to attach the replacement for ~BWCommon, looks like this.

Listing 8: Hello World from Pascal

```
{$keep "test"}

program test(output);

begin
writeln("Hello, world.");
end.

{$append "test.asm"}
```

The build script to automate the process of compiling and linking the various parts is almost a direct copy of the build script we used with ORCA/C.

Listing 9: Automating the Compile for Pascal

```
compile test.pas
assemble pasroot.asm
delete test.root
rename pasroot.root test.root
link test keep=test
test
```

A Clock Init

I really wanted to include a short but indispensable init as an example program. Instead, I settled for one that is merely a conversation piece. The sample init shown in Listing 1 sets up an interrupt handler that gets called 6 times a second. Each time it is called, it pokes the current time onto the text screen. If you are using a desktop application, you won't see a thing, but no harm will be done, either, since the text screen is reserved by the operating system. Any time you are using a text program, though, you will see the time at the top-right of your screen. As the screen scrolls, or new characters are placed on the screen by the text application, the time is written back to the original spot.

If you look at Listing 1, you will see that I renamed the assembly language file that contains ~_BWCommon. You should either rename yours or make a copy. I suppose you could change the #append directive, too, but I prefer keeping all of the files for a program together, so I made a copy of the file for this program.

I have used the clock with the text version of ORCA/M and the ORCA/M editor with no problems. I have also used it with several CDAs, again with no ill effects. In general, there shouldn't be any. It is possible, though, for a program to read the screen locations to get at stored text, rather than using a separate text buffer. If you have a program that does this, the time will be read by the program instead of the characters it placed there. I'm not aware of any Apple II programs that do this, but I would be surprised if there isn't at least one of them out there. If you run into a problem, of course, you can just delete the init from your SYSTEM.SETUP folder.

As with the simpler examples, creating the program is a bit involved. This is even more true with this init, since the file type has to be changed to $B6. The script is commented, so you should be able to follow it with no problems.

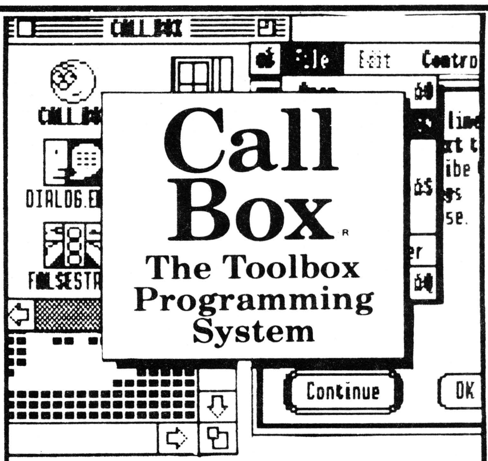
Listing 10: Building the Clock

```
* Compile the clock
compile clock.cc

* Replace clock.root with the rtl startup
code
assemble ccroot.asm
delete clock.root
rename ccroot.root clock.root

* link the executable
link clock keep=clock

* make it a permanent startup file
filetype clock $B6

* copy the program to the startup folder
copy -c clock 4/system.setup
```

There is one interesting point about the clock program that I would like you to notice. The clock is an interrupt driven program. Interrupt subroutines require a special header, and are called with 8 bit registers. The C compiler can't deal with either of these issues on its own, but a short assembly language patch handles the job very nicely. The patch, called HeartBeatTask, is a good example of when to use the mini-assembler built into C. The code is very short, and since it is written in C, you can move it from program to program, even if you aren't using the full-blown ORCA/M assembler.

Listing 11: The Clock Init

```
/
****************************************************************
*
*  Clock - screen clock
*
*  This program is an init.  It is executed as
*  your computer boots.  It installs a heartbeat
*  interrupt handler that writes the current
*  time to the top-right of the text screen.
*
*  By Mike Westerfield
*
*  Compiled under ORCA/C 1.0.
*
*  Source code released to the public domain.
*  The compiled code contains copyrighted
*  libraries from ORCA/C.
*
****************************************************************/


#pragma keep "clock"

#include <types.h>
#include <misctool.h>

/
****************************************************************
*
*  DrawClock - Draw a clock
*
*  This function writes the current time to the
*  top right of the text screen.  It does not
*  disturb the console drivers.
*
****************************************************************/


void DrawClock (void)

{
char *ptr0, *ptr1;  /* screen buffer pointers */
char time[20];      /* system time */

ReadAsciiTime(time); /* read the time */
ptr0 = (char *) 0x000424;/*set up screen ptrs */
ptr1 = (char *) 0x010424;
*ptr1++ = time[9]; /* place time on screen */
*ptr0++ = time[10];
*ptr1++ = time[11];
*ptr0++ = time[12];
*ptr1++ = time[13];
*ptr0++ = time[14];
*ptr1   = time[15];
*ptr0   = time[16];
}

/
****************************************************************
*
*  HeartBeatTask - Heartbeat interrupt task
*
*  This function implements interface required
* for a heartbeat interrupt handler.  Every
* 1/60th of a second, the system decrements
* count:.  When the value reaches 0, the
* code that follows is executed.  This code
* resets the timer and calls a normal C function
* to do the real work.
*
* Notes:  Inits tend to be small, so I took a
* short-cut by using phk-plb to set the data
* bank.  This sets the data bank to the value
* of the code bank, which works fine for the
* small memory model.  If you are using the
* large memory model, though, you will need to
* reset the data bank to ~GLOBALS, not
* to the code bank.  The C startup code shows
* how to do this.
*
*************************************************/


asm HeartBeatTask (void)

#define COUNT 10         /* heart beat counter */

{
        dcl      0    /* heartbt interrupt hdr */
count:  dcw      COUNT
        dcw      0xA55A

        phb         /* use our local data bank */
        phk
        plb
        php                 /* use long registers */
        rep      #0x30
        lda      #COUNT     /* reset task timer */
        sta      count
        jsl      DrawClock  /* call the C task */
        plp       /* switch to short registers */
        plb           /* return to the caller */
        rtl
}

#undef COUNT

/
*************************************************
*
*  main - main entry point
*
*  The main function is called at system startup
*  time.  It installs the heartbeat task and
*  returns.
*
*************************************************/


void main(void)

{
SetHeartBeat(HeartBeatTask);
}

#append "clock.asm"
```



Call Box®

The Toolbox Programming System

WYSIWYG?

(What You See Is What You Get)

Four powerful **WYSIWYG** editors slash programming time dramatically for Assembly, C, Pascal and Applesoft BASIC programs, YES! . . . I said Applesoft, CALL-BOX includes the first full function Applesoft BASIC interface for the IIgs toolbox as well but let's talk about the editors first.

- Image Editor . . .
  Create Icons, Cursors, and Pixel images in either 640 or 320 mode.
- Window Editor
  Create Window templates with scroll bars, controls, etc. plus custom colors.
- Dialog Editor . . .
  Create Dialog templates using Radio buttons, Check boxes, Line edit items, text in various styles, etc.
- Menu Editor . . .
  Create Menu templates with keypress equivalents, checks, diamonds, Font styles, etc.

All editors output APW source code, Linkable object code or resource files to make the best match to your current development system. Everything is accessable from the CALL-BOX Editor shell that includes these editors plus File utilities, Configuration utilities, programmable application launcher and the BASIC interface.

The CALL-BOX BASIC interface allows the Applesoft programmer to use Super Hi-Res via Quickdraw II, desktops, menu bars, windows, ports, fonts, dialog boxes, and the cursor linked task master system in the IIgs. This interface incorporates automated calls to minimize the code needed in your BASIC program and has added Long Call, Long Poke, Long Peek, and super array functions to bring Applesoft up to snuff with the additional memory in your IIgs.

All this plus a demo, sample code and bound manuals. Fully GS/OS V5.0 compatible and all in one place for the first time ever!

The CALL-BOX TPS **$99.00**
Add $4.50 shipping and handling.
Foreign add $10.50.
Send check, money order, Visa or MasterCard.

VISA

(714) 964-4298

SO WHAT SOFTWARE
10221 Slater Ave. #103, Fountain Valley, CA 92708

Apple II Infinitum

[Editor's note: II Infinitum is a campaign coordinated by Jerry Fellows to focus attention on the Apple II and to give Apple unmistakable proof that there is still much interest in the Apple II. Your letters to Apple and to the Wall Street Journal can make a difference in the future of the Apple II. The 8/16 editors fully support this campaign, just because it makes a lot of sense. = Jerry K. =)

February 1, 1990

To the Members of the Apple II Community:

This year could mark a historic turning point for the Apple II... if you help. We are asking you to voice your support for the Apple II, to convince Apple Computer that the Apple II is worth further investment.

Despite all the rumors regarding its imminent death, the Apple II remains with us, alive and improving. The Apple II community has, in many respects, been thrust backward into the days of semi-obscurity and grass-roots survival... however, Apple Computer is currently revitalizing its Apple II marketing and development strategies. With this effort comes the hope of a grand rebirth for the Apple II platform.

II Infinitum is a letter-writing campaign encouraging members of the Apple II community to speak out now! We want you to write not only to John Sculley at Apple Computer, Inc., but also to the Wall Street Journal. We hope that if the Journal receives enough letters, they will be motivated to publish an article on our efforts. This will allow us to then reach Apple stockholders, who have the clout that we need to support our efforts.

In addition, we urge you to distribute this letter to other members of the Apple II community, so that even more voices will be added to this cause. Listed on the following page are some guidelines that we recommend using when writing your letter. The addresses of John Sculley and the Wall Street Journal, as well as others we encourage you to contact, are listed after that.

Please take this opportunity to support the Apple II... only by combining our efforts can we achieve success.

Apple II Forever!
II Infinitum

Recommended Guidelines:

• Keep your letter businesslike and to the point - no more than one neatly typed or laser-printed page if possible.
• Avoid form letters or petitions; individual, personal letters have a much greater impact. Of course, you can write a single letter, then personalize it for each person you send it to.
• Include relevant personal information: perhaps discuss how long you have used the Apple II, the types of applications you use now or would like to use in the future, the direction you would like to see Apple take in developing, marketing and supporting the line, etc.
• Avoid negative or derogatory remarks. Focus on the positive and look toward the future.
• Be sure to close your letters by thanking the reader for his time.
• Mail your letters in a standard legal-size envelope which looks businesslike.
• Mail your letters with a return receipt requested if you can afford it.

Names and Addresses:

John Sculley
President and CEO
Apple Computer Inc
20525 Mariani Avenue
Cupertino CA 95014

Robert L Bartley Editor
The Wall Street Journal
200 Liberty Street
New York NY 10281

Letters
InCider Magazine
80 Elm Street
Peterborough NH 03458

Letters
Nibble Magazine
52 Domino Drive
Concord MA 01742

Letters Editor
Byte Magazine
One Phoenix Mill Lane
Peterborough NH 03458

The following are individuals at Apple Computer, Inc. to whom you may consider writing for greater effect (Write to them at the same address as John Sculley.):

Michael H. Spindler
Senior VP and President, Apple USA

Bernard Gifford
Vice President, Education, Apple USA

Randall S. Battat
Vice President, Product Marketing, Apple Products

David Hancock
Senior Vice President, Marketing, Apple USA

Morris Taradalsky
Vice President, Customer Service and Information Technology, Apple USA

Ian Diery
Senior Vice President, and President, Apple Pacific

Meet Other Apple II Developers!

See and hear about the latest Apple II hardware & software developments

Attend Apple's IIgs College

For most attendees, myself included, the Developers Conference hosted by ***A2-Central*** *in July 1989 was an experience bordering on the religious.*

Bill Kennedy, Technical Editor, *InCider*

Without exception, every attendee I have talked to feels the first ***A2-Central Developers Conference*** *at Avila College in Kansas City was a success. The retreat atmosphere was a significant factor in making it so.*

Cecil Fretwell, Technical Editor, *Call Apple*

As I look back, it was the most positive computer conference I have ever been to and I certainly recommend it to anyone with an interest in the Apple II line. Yes, I had a great time; yes, I learned a lot; yes, I met some outstanding people; and, yes, I'll go back.

Al Martin, Editor, *The Road Apple*

By popular demand, we're putting together another ***A2-Central Summer Conference*** (popularly known in developer circles as 'KansasFest'). Like last year, Apple is sending a number of its engineers to do seminars and to run a bug-busting room. Unlike last year, Apple is holding a IIgs College at Avila the day before our conference starts.

In addition to speakers from Apple, we'll have talks and demonstrations by active developers willing to show their tricks. There will be talks and exhibits by companies that provide tools to developers. And there will be plenty of time to talk to other developers.

You must register by June 1 to get the best prices, which begin at $300 and include all meals. For more information, call ***A2-Central*** at 913-469-6502 (voice), 913-469-6507 (fax) or write PO Box 11250, Overland Park, KS 66207. Or we're A2.CENTRAL on AppleLink and A2-CENTRAL on GEnie.

A2-Central Summer Conference
Avila College, Kansas City, Mo.
July 20 & 21, 1990

TIRED OF SWAPPING DISKS?

THEN *YOU* NEED A KAT HARD DRIVE!
BUILT *YOUR* WAY!

KAT hard drives come in industrial-quality cases that have, (115-230 volt) 60 watt power supplies, cooling fan, two 50-pin connectors and room for another half-height drive or tape back-up unit. Also included is a 6 ft. SCSI cable to go from the drive to your SCSI card. Now for the good stuff! You will also receive 20 meg of freeware, shareware, fonts, System 5.02 and public domain software.*Your* drive will have the interleave and partitions set for *You* before the drive is exercised for 24 hours. *You* get all of this and a one-year parts and labor warranty!

SB 48 Seagate 48 meg 40ms	**$549.99**
SB 85 Seagate 85 meg 28ms	**$698.99**
SB 105 Quantum 105 meg 12ms	**$899.99**

YOU BUILD IT!

SB CASE 2 HH Drives 7w 5h 16d	**$139.99**
ZF CASE 1 HH Drive 10w 3h 12d	**$169.99**
48 meg HD Seagate 40ms 3.5" SCSI	**$349.99**
85 meg HD Seagate 28ms 5.25" SCSI	**$469.99**
105 meg HD Quantum 12ms 3.5" SCSI	**$699.99**
T-60 TAPE Teac 60 meg SCSI	**$449.99**
W/ Hard Drive	**$424.99**
3.5" to 5.25" FRAME	**$12.50**
CABLE 25 pin to 50 pin 6 ft.	**$19.99**
50 pin to 50 pin 6 ft.	**$19.99**

NEW PRODUCTS!

VITESSE Inc. Salvation
Salvation is a slick new GS/OS-based volume backup/restore program for the IIGS. You can backup multiple, single or portions of large block devices including hard drives, RAM drives and ROM drives to 3.5" or 5.25" disks. Do you need to stop in the middle of the backup to get to an important file? No problem with Salvation. It remembers where you left off and starts back up at that point. Uses the familiar Apple Desktop Interface. **$39.99**

QUICKIE
Quickie is the hand-held scanner we've all been waiting for! You get up to 400 DPI and 16 shades of gray. Watch the image apear on the screen as you scan then import it into your favorite paint, draw or graphics program. **$249.99**

COMPUTER PERIPHERALS ViVa24
The ViVa24 is a 2400 baud modem that is 100% Hayes compatible. Unique "tower" design allows for better viewing of the status icons used in place of cryptic LED's on some modems. Comes with a FIVE - YEAR WARRANTY! **$139.99**

HARRIS LABORATOIES, Inc. GS Sauce
The GS Sauce is a compact memory board that differs from most of the rest. It uses low-power, cool-running CMOS SIMMs like the Mac. You can use 256K or 1 meg SIMMs for a total of 4 megs. Made in the USA. Limited lifetime warranty. **$79.99**

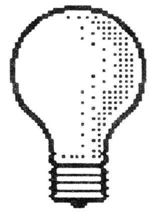
SOFTWARE, ACCS ETC.

1 meg SIMMs 80 ns	**$89.99**
1 meg x 1 80 ns	**8 / $79.99**
Æ Conserver	**$79.99**
Æ Transwarp GS	**$289.99**
AI Juice Plus W/1 meg	**$144.99**
CH PRODUCTS FLIGHT STICK	**$49.99**
KENSINGTON SYSTEM SAVER GS	**$69.99**
KENSINGTON TURBO MOUSE ADB	**119.99**
KEYTRONIC KEYBOARK 105 KEYS ADB	**$139.99**
BYTE WORKS ORCA/C	**$89.99**
BYTE WORKS ORCA/M	**$44.99**
BYTE WORKS ORCA/PASCAL	**$89.99**
BYTE WORKS DISASSEMBLER	**$34.99**
CHECKMATE PROTERM 2.1	**$89.99**
ROGER WAGNER HYPERSTUDIO	**$94.99**
ROGER WAGNER MACROMATE	**$37.99**
STONE EDGE DB MASTER PRO	**$219.99**
GENERIC 3.5" DS/DD BULK	**50 / $.69**

KAT

Phone: (913) 642-4611
Or Mail Orders To: **KAT**
8423 W 89th Street
Overland Park, KS 66212-3039

Gimme a Light



by Jerry Kindall, Classic Apple Editor

LIGHT is a simple line editor I wrote to assist me in the editing of BASIC programs. Its major advantage is that it fits entirely into page 3 of RAM, which means that it doesn't take any program space away from BASIC, allowing you to edit those really tight programs. Of course, with only 192 bytes of code, its editing capabilities are rather rudimentary, but LIGHT has more features than you might expect.

LIGHT has character insert and delete, and control-character override to allow you to enter even illegal characters into a line. It runs under both DOS 3.3 and ProDOS and will work on any machine from an Apple II+ to a IIgs, or even a clone. It even has some rudimentary 80-column support, and works in Applesoft, the Monitor, or even the mini-assembler. All this in less than 192 bytes!

This article is really two articles in one. The first part is intended for Applesoft programmers who just want to use LIGHT to make quick and dirty changes to programs. The second part may be of interest to assembly-language programmers, as it shows how to fit a maximum of functionality into a minimum of code; it begins under the subhead "How It Works".

Turning On The LIGHT

To install LIGHT, just BRUN it, and it will connect itself to the ampersand hook. Once you have installed LIGHT, it lies dormant waiting for you to issue an ampersand command. When it sees an ampersand, LIGHT connects itself to the BASIC I/O hooks to intercept your keystrokes.

LIGHT Switches

After LIGHT has been connected by an ampersand command, the following four keys become LIGHT command keys:

Tab (Control-I on Apple II+): Insert blank space at cursor

Delete (Control-D on II+): Delete character left of cursor

Control-O: Enter control character into line

Control-X: Move cursor to first character of input line

The Insert and Delete command keys work differently from most other line editors, such as the venerable GPLE. Instead of moving everything to the right of the cursor forward and backward, Insert and Delete work with the stuff to the left of the cursor. LIGHT isn't really a line editor in the strictest sense of the word, it's just a supplement to the Apple's built-in line editor (such as it is), and the built-in line editor only keeps track of characters to the left of the cursor. It's a disconcerting effect at first, but it works.

The Control-O (Override) feature does not automatically insert a space for the character entered. You'll have to do that ahead of time with Insert. The Control-X command replaces the Apple's normal cancel line command; the new Control-X is functionally equivalent to the old one, since moving the cursor to the beginning of the line causes the Apple's built-in editor to forget everything you've typed. This one's just cleaner, that's all.

Try it out! That's the best way to get used to LIGHT's handy features. Remember, once you activate it with the ampersand command, it's always active, so you can hit a LIGHT editing key at any time.

Editing Existing Lines

To edit a line that's already part of your program, LIST it on the screen. Then, using the usual Escape commands (Escape followed by the arrows or the IJKM diamond), move the cursor to the first digit of the line's line number. Then press ESC again to exit cursor-moving mode. Now use the right arrow key to move to your first mistake, and use the Insert and Delete commands to fix it. Use the left and right arrow keys to move throughout the line, editing as needed. When you are done, trace over the rest of the line with the right arrow key before pressing Return.

Compressed Listings

To edit a REM or DATA statement, you can use the command POKE 33,33, which will stop Applesoft from indenting its listings. This is an old trick and isn't specific to LIGHT. You could also use LIGHT's Delete command to delete unwanted spaces; remember, Applesoft adds an extra space (which should be deleted) after the REM or DATA token.

LIGHT also has a command designed especially for compressing listings. Simply follow the ampersand with the number of the line to list. LIGHT performs a POKE 33,33 to cancel indentation and also removes all spaces from a listing, except those inside quotation marks. This dense-pack text display is ideal for editing lengthy program lines.

To return to full-screen editing, type TEXT.

LIGHT Up Control Characters

You may have noticed that LIGHT displays most control characters as inverse letters. When when you trace over an inversed control character, LIGHT will pick it up as if you'd typed it on the keyboard. The Return (Control-M), Backspace (Control-H), and Bell (Control-G) characters, however, are printed normally during LISTs to preserve normal screen formatting.

If you enter one of the three special characters using Control-O, it will be displayed as an inverse M, H, or G, just as it should be. If you try to edit a line containing these control characters, though, you'll lose them, because they aren't displayed as inverse letters during LISTing.

Double The LIGHT In 80-Column Mode

Some of LIGHT's features also work in 80-column mode on the enhanced IIe, the IIc, and the IIgs, but not the original IIe or the II+. The compressed listing command (& line-number) works, and even sets a 72-column screen window. The Insert, Delete, and Control-X commands work fine as long as there are no control characters to the left of the cursor. The control-O command does not work, and neither does the inverse control-character feature.

Switching to or from 80-column mode will disconnect LIGHT. You should issue the & command after switching to reconnect it.

Turning Off The Light

When LIGHT is connected, it will respond to its keyboard commands anytime you see a cursor on the screen, even during GET and INPUT statements in BASIC programs. This usually isn't what you want, so you should disconnect LIGHT before running your program. Disconnection is vital if your program uses page 3 of memory, as many programs do; overwriting LIGHT while it is still connected will cause crashes. If you do overwrite LIGHT with another program, you must BRUN it from disk again to install it.

The easiest way to disconnect LIGHT is to reset the computer, which restores standard I/O hooks, as well as a full-screen text window, canceling the effects of POKE 33,33 or LIGHT's compressed-lister command.

Ampersand-less LIGHT

If you want to use the ampersand hook for another utility, be sure to install LIGHT before installing the other program, because LIGHT does not pass on unrecognized ampersand commands to other ampersand utilities. If this is not possible, or if the other program also does not pass on ampersand commands, you can BLOAD LIGHT (not BRUN). If you're running under DOS 3.3 (say what?) you will also need to CALL 714 after BLOADing LIGHT.

After loading LIGHT in this manner, you can use CALL 771 in place of an ampersand call with no parameters to connect LIGHT. You can use CALL 768,num in place of an ampersand call followed by a line number to list a line in compressed format. The comma between the 768 and the line number is required.

You LIGHT Up My Program

If you are not using page 3 for another utility, you can use LIGHT's editing features in your BASIC programs. BLOAD LIGHT near the beginning of the program. Just before your INPUT statement, CALL 771 to connect LIGHT. The user of your program will be able to use LIGHT to edit their input. After the INPUT, disconnect LIGHT using PRINT CHR$(4);"PR#0": PRINT CHR$(4);"IN#0". Do not leave LIGHT connected during GET statements, or during INPUT from disk.

How It Works

The main BRUN entry is at lines 32-59 and resides in the keyboard buffer, since it is not needed after execution. This section of code has three tasks: first, it connects LIGHT to the ampersand vector (since LIGHT's ampersand entry is at $303, I can just store the same value into both bytes of the ampersand vector). Second, LIGHT checks to see what operating system it's running under. If it's DOS 3.3, the program is modified to use the DOS 3.3 I/O hooks at $AA53-$AA46 instead of the ProDOS I/O hooks. Finally, if the computer is running on a II+, the check for an 80-column display is disabled and the check for the Delete key is changed to look for a Control-D instead.

Lines 63-91 are the main CALL and ampersand entry point. If the program is entered with CALL 768, a call to chkcom is made to check for the comma before the line number. We set up the I/O vectors to point to our special I/O routines, and set up flags so that all unquoted spaces will be removed from the output stream. Next we check the character after the ampersand or call to see if it's numeric. If it is, we set a 33 (or 73) column window and exit through Applesoft's LIST routine to list the line on the screen. Otherwise we deactivate space filtering and simply return to BASIC.

When the Apple wants a keypress, the input routine in lines 95-108 is called. This routine calls keyin to get a keypress, then checks for each of our new command keys. The Control-X command is handled by lines 106 and 107, which simply backspace to the start of the input line and go get another keypress.

The Delete command is handled in lines 112-120 by backspacing to the beginning of the input line, printing a space, and then reprinting all but the last character of the input line. This shifts everything to the left of the cursor one space to the right, leaving the cursor in the same place but deleting the character to the left of the cursor.

The Insert command (lines 124-131) works similarly, backspacing one beyond the beginning of the input line and moving everything to the left of the cursor back a space. The cursor moves left along with the text, leaving space to type new characters.

Lines 135-142 allow the user to enter any control character after pressing Control-O. The cursor freezes, and a keypress is accepted and placed directly into the buffer and onto the screen.

The back, linout, and outdo subroutines at lines 144-175 are called by the Insert and Delete routines. Back moves the cursor to the beginning of the input line. Linout prints the output line from the beginning to the current cursor position. Outdo prints the current character in inverse if it's a control character, or normally if not.

The output routine (lines 179-202) is called whenever Applesoft wants to print a character. It is this routine which filters spaces from the Applesoft listing, and prints Return, Backspace, and Bell normally to preserve screen formatting.

To keep the code size down I used what is commonly known as spaghetti code: lots of wierd branches around, one rts serving several subroutines, and things like that. I also used self-modifying code in the setup routine. In short, I did a number of things that you're not supposed to do, but the benefit is that the code is the smallest possible size and actually packs quite a wallop. If ever you have need to write super-compact code, LIGHT can serve as an example.

You could also use LIGHT as a starting point for a more sophisticated editor. If you gave yourself a few more bytes, say a total of 256, or 512, you ought to be able to add a few new editing commands and make the compact listing more flexible. By using 65C02 opcodes you could fit even more power into your limited space. Anyway, I hope you enjoy it!

LIGHT Program Listing

```
                 *********************************
                 *                               *
                 * LIGHT                         *
                 * by Jerry Kindall              *
                 *                               *
                 * Merlin 8 Assembler            *
                 *                               *
                 *********************************

                 indflg   =     $04        ;index and quote flag
                 filter   =     $05        ;filter spaces from output?
                 wndwdth  =     $21        ;text window width
                 chrgot   =     $B7        ;get char at TXTPTR
                 buf      =     $200       ;keyboard buffer
                 doswrm   =     $3D0       ;DOS warm start vector
                 amper    =     $3F5       ;& vector
                 pcsw     =     $BE30      ;ProDOS output hook
                 pksw     =     $BE32      ;ProDOS input hook
                 rd80col  =     $C01F      ;bit 7 hi if 80-cols on
                 list     =     $D6A5      ;BASIC list routine
                 outspc   =     $DB57      ;print a space
                 chkcom   =     $DEBE      ;skip over comma at TXTPTR
                 back1    =     $FC10      ;backspace once
                 rdkey    =     $FD0C      ;input from current device
                 keyin    =     $FD1B      ;input from keyboard
                 cout1    =     $FDF0      ;output to screen

                          org   $2BD

                 * Main BRUN entry:

02BD: A9 4C               lda   #$4C       ;set up & vector
02BF: 8D F5 0             sta   amper
02C2: A9 03               lda   #start     ;low byte & hi byte of
02C4: 8D F6 03            sta   amper+1    ; entry point are the same
02C7: 8D F7 03            sta   amper+2
02CA: AD D1 03            lda   doswrm+1   ;check for DOS/ProDOS
02CD: F0 1F               beq   :2         ;ProDOS - program OK
02CF: A2 AA               ldx   #$AA       ;otherwise modify for
02D1: 8E 07 03            stx   mod1+2     ; DOS 3.3 use: DOS I/O
02D4: 8E 0C 03            stx   mod2+2     ; hooks are on page $AA
02D7: 8E 11 03            stx   mod3+2
02DA: 8E 14 03            stx   mod4+2
02DD: A2 53               ldx   #$53       ; $AA53-$AA56 for I/O hooks
02DF: 8E 0B 03            stx   mod2+1
02E2: E8                  inx
02E3: 8E 10 03            stx   mod3+1
02E6: E8                  inx
02E7: 8E 06 03            stx   mod1+1
02EA: E8                  inx
02EB: 8E 13 03            stx   mod4+1
02EE: AD B3 FB   :2       lda   $FBB3      ;is it Apple II+?
02F1: C9 EA               cmp   #$EA       ;no ‾ program OK
02F3: D0 0A               bne   :3
02F5: A9 21               lda   #33        ;otherwise disable 80-col
```

```
02F7: 8D 29 03              sta    L1+1
02FA: A9 84                 lda    #$84         ;and change delete key to
02FC: 8D 39 03              sta    L4+1         ; ctrl-D
02FF: 60          :3        rts                 ;exit until later

                  * Ampersand/CALL entry

0300: 20 BE DE              jsr    chkcom       ;entry for CALL 768,##

0303: A9 33       start     lda    #input       ;entry from ampersand call
0305: 8D 32 BE    mod1      sta    pksw         ; set up I/O vectors to
0308: A9 A5                 lda    #output      ; point to output and input
030A: 8D 30 BE    mod2      sta    pcsw
030D: A9 03                 lda    #/input
030F: 8D 33 BE    mod3      sta    pksw+1
0312: 8D 31 BE    mod4      sta    pcsw+1

0315: 85 04                 sta    indflg       ;clear hi bit of both
0317: 85 05                 sta    filter       ; filter and indflg
                                                ; turning on space filter
0319: 20 B7 00              jsr    chrgot       ;get char after &
031C: B0 12                 bcs    L3           ;if not # then return to BASIC

031E: 08                    php                 ;save processor status
031F: 2C 1F C0              bit    rd80col      ;80-columns on?
0322: 30 04                 bmi    L1           ;yes, use 72-col window
0324: A2 21                 ldx    #33          ;otherwise use 33 cols
0326: D0 02                 bne    L2
0328: A2 48       L1        ldx    #72          ;changed to ldx #33 on II+
032A: 86 21       L2        stx    wndwdth
032C: 28                    plp                 ;get status flags back

032D: 4C A5 D6              jmp    list         ;& enter BASIC list

0330: 66 05       L3        ror    filter       ;set hi bit of filter
0332: 60                    rts                 ; to deactivate space strip

                  * Keyboard input entry

0333: 20 1B FD    input     jsr    keyin        ;get a keypress
0336: 85 05                 sta    filter       ;turn off output filter

0338: C9 FF       L4        cmp    #$FF         ;delete (changed to ^D on II+)
033A: F0 11                 beq    delete       ; so delete char
033C: C9 89                 cmp    #$89         ;control-I (Tab)
033E: F0 1E                 beq    insert       ; so insert char
0340: C9 8F                 cmp    #$8F         ;control-O
0342: F0 2C                 beq    ctrl         ; so enter ctrl char
0344: C9 98                 cmp    #$98         ;control-X
0346: D0 42                 bne    backx        ; handle it right here
0348: 20 7E 03              jsr    back         ;go to beginning of line
034B: F0 20                 beq    rd           ;(always) get next char
```

```
                                               ;x-reg is zero now

                * Delete char to left of cursor

034D: E0 00     delete    cpx   #0             ;if at first char pos,
034F: F0 1C               beq   rd             ; nothing to delete

0351: 20 7E 03            jsr   back           ;move cursor back
0354: 20 57 DB            jsr   outspc         ;print a space
0357: C6 04               dec   indflg         ;delete char from buffer

0359: 20 8B 03            jsr   linout         ;print buffer contents
035C: F0 0F               beq   rd             ;always

                * Insert blank at cursor

035E: 20 7E 03  insert    jsr   back           ;back to beginning
0361: 20 10 FC            jsr   back1          ;back one more

0364: 20 8B 03            jsr   linout         ;then print buffer
0367: 20 57 DB            jsr   outspc         ;and a space
036A: 20 10 FC            jsr   back1          ;then a backspace

036D: 4C 0C FD  rd        jmp   rdkey

                * Enter control character

0370: A9 FF     ctrl      lda   #$FF           ;freeze cursor
0372: 20 1B FD            jsr   keyin          ; and get character

0375: 20 9A 03            jsr   outdo          ;now output it

0378: 9D 00 02            sta   buf,x          ;put it in buffer
037B: E8                  inx                  ;move cursor right 1
037C: D0 EF               bne   rd             ;always

                * Backspace to start of input

037E: 86 04     back      stx   indflg         ;save x register

0380: E0 00               cpx   #0             ;if no characters,
0382: F0 06               beq   backx          ; do nothing

0384: 20 10 FC  :1        jsr   back1          ;backspace
0387: CA                  dex
0388: D0 FA               bne   :1

038A: 60        backx     rts

                * Output contents of input buffer

038B: A2 00     linout    ldx   #0             ;beginning of buffer
```

```

038D: E4 04     :1        cpx   indflg      ;are we at end?
038F: F0 F9               beq   backx       ;yes ¯ exit

0391: BD 00 02            lda   buf,x       ;no - print char
0394: 20 9A 03            jsr   outdo
0397: E8                  inx
0398: D0 F3               bne   :1          ;always

                * Output control characters in inverse

039A: 29 7F     outdo     and   #$7F        ;clear hi bit
039C: C9 20               cmp   #$20        ;control character?
039E: 90 02               bcc   docout      ;yes ¯ print inverse
03A0: 09 80               ora   #$80        ;otherwise restore hi bit
03A2: 4C F0 FD  docout    jmp   cout1

                * Output while filtering spaces

03A5: 24 05     output    bit   filter      ;should we remove spaces?
03A7: 30 14               bmi   :2          ;no - output normally

03A9: C9 A2               cmp   #$A2        ;got quote mark?
03AB: D0 08               bne   :1          ;no

03AD: A5 04               lda   indflg      ;yes - toggle quote flag
03AF: 49 80               eor   #$80
03B1: 85 04               sta   indflg
03B3: A9 A2               lda   #$A2        ;and restore quote char

03B5: C9 A0     :1        cmp   #$A0        ;do we have a space?
03B7: D0 04               bne   :2          ;no

03B9: 24 04               bit   indflg      ;yes, is it in quotes?
03BB: 10 CD               bpl   backx       ;no - exit without printing

03BD: C9 8D     :2        cmp   #$8D        ;is char CR?
03BF: F0 E1               beq   docout      ;yes - print thru cout1
03C1: C9 88               cmp   #$88        ;is it BS?
03C3: F0 DD               beq   docout      ;print thru cout1
03C5: C9 87               cmp   #$87        ;is it BELL?
03C7: F0 D9               beq   docout      ;so beep already!
03C9: D0 CF               bne   outdo       ;always, print ctrl char inverse

¯End assembly, 270 bytes, Errors: 0
```

The Merlin Maniac

Rolling Your Own (Controls)

by Steve Stephenson

I was working on a project recently that needed buttons in the window. It is a pain to juggle controls and scrolling text in the content area of a window (without the new Text Edit). So, I decided that I would put the controls in the Info Bar. I had read GS Tech Note #3 which hinted that this could be done. After many frustrating hours of trying to make it work, I discovered that the Tech Note had been revised. It says, "(Note: The Control Manager currently will not allow controls it creates in an information bar. In this case, NewControl would be using a port that is not in your window's port, namely the Window Manager's port.)". I decided that since I couldn't use the Control Manager, I would put together some routines that would look and act the same.

The parts of this project fall into three categories: - Creating and updating the info bar itself. - Detecting and responding to info bar events. - Other routines to simulate the Control Manager.

Creating and Updating

Creating an info bar seems simple enough; however, there are some 'gotchas'. The first one that seems to get everyone is that your update routine (to redraw the info bar) gets called DURING _NewWindow! If your routine uses things that are not ready until after the window is established, you will probably see just the hollow window frame drawn as your pride and joy expresses its frustration with the customary 'bonk'!

I was also thrown by the coordinate system that is used in the info bar. Where the 0,0 point for everything else you put in a window is the upper left corner of the content area; the 0,0 point for the info bar is the upper left corner of the window's frame! So if you start drawing at 0,0, you won't see anything. Even if you move down by a unit of the font height, you still won't be far enough. In the listing, you will find that I used a constant, "InfoBarTop", that is the height of the title bar that has to be added to get down to the real top of the info bar.

To coax your window to show an info bar, you need to set the 'fInfo' frame bit. You also set the number of pixels tall the bar needs to be. That's easy; but now you need a procedure to draw the inside of the bar. It's one of those strange procedures where vital variables are already on the stack for you, but you get to pull them off when you're done.

My update routine ("InfoUpdate") just draws 4 'buttons' in their proper state. It should be easy enough to follow, but there are some items that may need a little explanation. For example, my choice of a 3 by 1 pensize (rather than the standard 1 by 1) seemed to me to look most like what the Control Mgr uses.

I used rounded rectangles because the rest of the program was using them; it certainly would have been easier to draw regular rectangles! If you're curious about "OvalHeight" and "OvalWidth", they are required by _FrameRRect, _EraseRRect, and InvertRRect. How did I come up with these values? Well, after a lot of thought and attempts to create a formula, I was unable to find a correlation between the values needed and the rest of the rectangle, so the values are the result of some tedious trial and error.

One other item that could stand a little light is my use of the tables, "OutsideRects" and "InsideRects". I have always found it tedious to construct a table of numbers, and a much bigger pain to make changes. The ultimate pain comes when you try to update your program months or years later. So, when I need to construct a list of values such as this, I try to boil it down to the few items that I might need to change in the future and assign them as constants. Then create a table entry that is based on those constants. With a little care and planning, you can loop for the total number of items and let Merlin generate the table for you.

My button titles also needed to be able to change, so the update routine allows for varying title string lengths and automatically centers the string.

I also had to be able to alternate the buttons between enabled and disabled. The status of each button is kept in the "EnableTable". To show a button as disabled, you first draw it normally, then erase every other pixel.

Detecting and Responding

The central core of detecting a hit in a button uses _PtInRect to compare whether the point of the mouse click is within the area of the button. To simulate a _FindControl, we loop through all four buttons checking the point. But first, we must get the coordinate systems on the same level. A call to _StartInfoDrawing will set the coordinates relative to the info bar (this call must be balanced by an _EndInfoDrawing call). Then _GlobalToLocal will convert the point; the 'local' is now the info bar. We also need to supply _StartInfoDrawing with the pointer to the info bar's RECT. This RECT is found with _GetRectInfo and only needs to be done once; a good place to put the call is right after creating the window.

When "CheckHit" returns the variable, "inButton" set to True, it's time to 'track' the control. To do this, we set up a loop that continues while the mouse button is still down. Each time through, it checks the location of the mouse. Just like the real _TrackControl, a release of the button when out of the control is not considered a hit. So, every time the mouse strays out of the control, it is inverted back to normal.

When the mouse button is released, "ButtonNum" holds the local number of the hit (a miss is assigned the number zero). It is then a simple matter of looking up the address of that button's handler. I did not provide any useful handlers for these buttons as that is entirely up to you and what your program needs.

Other Routines

I've thrown in some other routines that you may need to complete the simulation of the Control Manager.

To change the state of a button's enabling, just change it's entry in the "EnableTable" and call _DrawInfoBar. This call redraws the entire info bar using your update routine. Refer to "DisableButton".

To change the button's title, I provided the routine "ChangeButtonTitle", which clears the current button's rectangle and calls the low level routine "DrawButton" to redraw it. See "DoButton3" for an example.

You might like to have your buttons also respond to key equivalents. The example, "HotKey", shows what to do after detecting a key event and deciding that it is yours to handle. It uses the low level routine "SelectButton" to flash the button on and off, then uses the low level entry point "GoButton" to be handled by that button's routine.

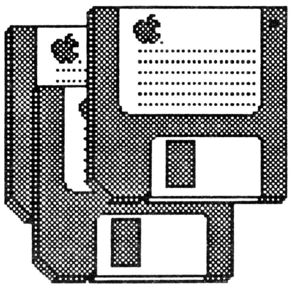
```
*===============================================
*Copyright 1990 Steve Stephenson & Ariel Pub
*Some rights reserved.

* some constants

top       =     0           ;rect offsets
left      =     2
bottom    =     4
right     =     6

oWhat     =     0          ;event record offsets
oMessage  =     2
oWhen     =     6
oWhere    =     10
oModifiers =    14

active    =     $00         ;boolean constants
inactive =      $ff

InfoBarTop =    13          ;offset to bar top
spacing   =     10          ;between buttons
buttonwidth =   80          ;width of a button
buttonheight =  11          ;height of a butto
ovalWidth =     24
ovalHeight =    8

WindowPtr ext               ;you provide these
EventRecord ext

*===============================================
InfoUpdate ent              ;Window Mgr only!
          phb               ;save B
          phk               ;reset B
          plb
          phd               ;save D
          tsc               ;reset D
          tcd
* what the dpage-in-stack looks like:
          dum   1           ;stk ptr
:d        ds    2           ;saved D
:b        ds    1           ;saved B
:rtl      ds    3           ;caller's rtn addr
:windPtr adrl   0           ;window's port
```

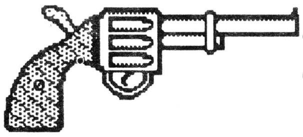
```
:iRefCon adrl  0             ;infobar RefCon
:iRect   adrl  0             ;infobar¬s RECT
         dend

         ~GetPenMask #origmask ;save mask
         ~GetPenSize #origsize ; & pensize

         ~SetPenSize #3;#1 ;reset pen
         lda   #4            ;# of buttons
         sta   ButtonNum
:drawlp  jsr   DrawButton ;make one button
         dec   ButtonNum;countdown til done
         bne   :drawlp

         ~SetPenSize #origsize;#origsize+2

         pld                 ;restore D
* pull B & the RTL addr off temporarily
         plx                 ; B & rtl bnk
         ply                 ; rtl addr
* pop the stuff that was passed to us
         pla                 ; (windowptr)
         pla
         pla                 ; (refcon)
         pla
         pla                 ; (rect)
         pla
* now put the B & RTL addr back onto stk
         phy                 ; rtl addr
         phx                 ; rtl bnk & B
* and exit to caller
         plb                 ;restore B
         rtl                ;back to Window Mgr

origmask ds    8
ButtonNum dw   0
*
* Draws one complete button (ButtonNum set)
DrawButton
         jsr   GetRectOffset ;table index

* draw the frame
         pea   #^OutsideRects ;push hi word
         lda   #OutsideRects;start of table
         clc
         adc   RectOffset;+ button's offset
         pha                 ; push lo word
         pea   #ovalWidth ;oval dimensions
         pea   #ovalHeight
         _FrameRRect         ;draw the outside

* lookup the ptr to the title string
         pea   #^Titles ;hi word (all same)
         lda   ButtonNum
         asl
         tax
         lda   Titles-2,x ;lo word
         pha                 ;(for _DrawString)

* find width of str for centering title
         pha                 ;space
         pea   #^Titles      ;calc title width
         pha                 ; for centering
         _StringWidth
                             ; (width on stk)
* position the pen for drawing the title
         lda   #buttonwidth ;width of button
         sec
         sbc   1,s           ;minus title width
         lsr                 ;div 2 = offset
         sta   1,s           ;  from left
         ldx   RectOffset
         lda   OutsideRects+left,x;left side
         adc   1,s           ;+ offset
         sta   1,s           ;= horiz start
         pea   #InfoBarTop+10;move vert down
         _MoveTo             ;  by font height

* now draw it (title ptr still on stk)
         _DrawString

* set the dimming as required
         lda   ButtonNum
         asl
         tax
         lda   EnableTable-2,x ;enabled?
         beq   :active       ; yes, leave as is
:inactive                    ; no, draw disabled
         ~SetPenMask #dimmask
         pea   #^InsideRects;use inside rect
         lda   #InsideRects
         clc
         adc   RectOffset
         pha
         pea   #ovalWidth-3 ;& reduced ovals
         pea   #ovalHeight-3
         _EraseRRect         ;thru the dim mask
         ~SetPenMask #origmask;& restore pen
:active
         rts

*
* Calc the offset into the table of rects
GetRectOffset
         lda   ButtonNum
         dec                 ;(make 0-relative)
         asl                ;*8 bytes in a rect
         asl
         asl
         sta   RectOffset
         rts

RectOffset dw  0
*
```

```
* Invert the inside of the button
* RectOffset must already be setup
InvertButton
         lda   black      ;flip the state
         eor   #1         ; of the color
         sta   black      ;(for "track ctl")

         pea   #^InsideRects
         lda   #InsideRects ;use inside rect
         clc
         adc   RectOffset
         pha
         pea   #ovalWidth-3 ;& reduced ovals
         pea   #ovalHeight-3
         _InvertRRect     ;reverse it
         rts

black    dw    0

*==========================================
* Call here on a mouse click in the info bar
InInfoBar ent
         lda   EventRecord+oWhere+2
         sta   thePoint+2 ;copy of the point
         lda   EventRecord+oWhere
         sta   thePoint

         jsr   StartInfoDraw ;change coords
         ~GlobalToLocal #thePoint;convert pt

         lda   #4         ;loop thru each
         sta   ButtonNum
:find    lda   ButtonNum
         asl
         tax
         lda   EnableTable-2,x ;enabled?
         bne   :disabled  ; no, skip it
         jsr   GetRectOffset
         jsr   CheckHit   ; yes, hit here?
         bne   :hit       ; yes, handle it
:disabled                 ; no, next button
         dec   ButtonNum  ;done all buttons?
         bne   :find      ; no, loop
         _SysBeep         ; yes, beep
         bra   :done

* do "Track control"
:hit                      ;track the hit
         lda   black      ;is it inverted?
         bne   :stilldown ; yes, skip invert
         jsr   InvertButton ;no, go to black
:stilldown
         ~StillDown #0    ;see if holding btn
         pla
         sta   mouseDown
         beq   :up        ;btn released
         ~GetMouse #thePoint;holding, is loc
         jsr   CheckHit   ; still in button?
         bne   :hit       ;  yes, continue

* button either released or mouse strayed
:up
         lda   black      ;already outside?
         beq   :stray?    ; yes, skip invert
         jsr   InvertButton ;no, go to white
:stray?
         lda   mouseDown  ;still holding?
         bne   :stilldown ; yes, track it
         lda   inButton   ; no, released
         bne   :done      ; good hit, handle
         stz   ButtonNum  ; outside, ignore
:done
         jsr   EndInfoDraw ;reset coords

GoButton ent              ;handle the hit
         lda   ButtonNum
         beq   :none      ;strayed, ignore it
         asl
         tax
         jsr   (Buttons-2,x) ;do the fn
:none    rts

Buttons                   ;to your routines..
         da    DoButton1
         da    DoButton2
         da    DoButton3
         da    DoButton4

mouseDown dw   0
thePoint dw    0,0
*
* See if click is in the button
* Enter with RectOffset already calculated
* Returns boolean result in inButton
CheckHit
         pha              ;space for result
         PushLong #thePoint
         pea   #^InsideRects
         lda   #InsideRects ;use inside rect
         clc
         adc   RectOffset
         pha
         _PtInRect        ;hit in the rect?
         PullWord inButton ; result
         rts

inButton dw    0

*==========================================
* Reset coordinates relative to the info bar
* Save original & reset pensize
StartInfoDraw
         ~StartInfoDrawing #iRect;WindowPtr
         ~GetPenSize #origsize ;save
```

```
            ~SetPenSize #3;#1 ;reset
            rts

iRect       ent
            dw    0,0,0,0
origsize    dw    0,0

*===========================================
* Restore pensize to original
* Reset coordinates back to window
EndInfoDraw
            ~SetPenSize #origsize;#origsize+2
            _EndInfoDrawing
            rts


*===========================================
* Flash the ButtonNum button on and off
SelectButton ent
            jsr   StartInfoDraw
            jsr   GetRectOffset
            jsr   InvertButton ;hilite on
            ldx   #$8000      ;short pause
:delay      dex
            bne   :delay
            jsr   InvertButton ;hilite off
            jmp   EndInfoDraw


*===========================================
* Call here after installing new title ptr
*  in the Titles table; redraws the button
ChangeButtonTitle
            jsr   StartInfoDraw
            jsr   GetRectOffset
            pea   #^OutsideRects ;clr old rect
            lda   #OutsideRects
            clc
            adc   RectOffset
            pha
            pea   #ovalWidth
            pea   #ovalHeight
            _EraseRRect
            jsr   DrawButton ; with new title
            jmp   EndInfoDraw


*===========================================
Titles
            da    str:b1
            da    str:b2
str:ptr     da    str:b3
            da    str:b4

str:b1      str   'More'
str:b2      str   'Less'
str:b3      str   'Maybe'
str:b3:a    str   'Why not'
str:b4      str   'Really?'

EnableTable ent
            dw    inactive
            dw    inactive
            dw    active
            dw    active

OutsideRects
]right      =     0          ;init to 0
            lup   4          ;make 4 outer rects
]top        =     InfoBarTop+1
]left       =     ]right+spacing
]bottom     =     ]top+buttonheight
]right      =     ]left+buttonwidth
            dw    ]top
            dw    ]left
            dw    ]bottom
            dw    ]right
            --^


InsideRects
]right      =     0          ;init to 0
            lup   4          ;make 4 inner rects
]top        =     InfoBarTop+1
]left       =     ]right+spacing
]bottom     =     ]top+buttonheight
]right      =     ]left+buttonwidth
            dw    ]top+1      ;inset from frame
            dw    ]left+3     ; 1 high & 3 wide
            dw    ]bottom-1
            dw    ]right-3
            --^


DimMask
            dfb   %01010101
            dfb   %10101010
            dfb   %01010101
            dfb   %10101010
            dfb   %01010101
            dfb   %10101010
            dfb   %01010101
            dfb   %10101010


*===========================================
* The Button Handler routines

*===========================================
DoButton1
DoButton2
DoButton4
                                 ;it's up to you...
            rts

*--------------------
```

```
DoButton3                       ;swap to alt title
          lda     titleFlip
          eor     #1            ;toggle state
          sta     titleFlip
          bne     :alt
          lda     #str:b3       ;prime string
          bra     :flip
 :alt     lda     #str:b3:a     ;alternate string
 :flip
          ldx     #b3*2
          sta     Titles-2,x ;install new ptr
          jsr     ChangeButtonTitle;& redraw it

 * then do whatever you need to...
          rts

 titleFlip dw     0

*==========================================
 HotKey   ent
          sta     ButtonNum   ;lookup in table
          asl
          tax
          lda     EnableTable-2,x ;enabled?
          bne     :disabled   ; no, skip it
          jsr     SelectButton ; yes, flash it
          jmp     GoButton    ; & handle it.

 :disabled
          rts


*==========================================
 DisableButton
          ldx     #b2*2        ;dim 2nd button
          lda     #inactive
          sta     EnableTable,x
          ~DrawInfoBar WindowPtr
          rts


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





Hired Guns

8/16 is providing a free service to all programmers (who are subscribers!): placement of a complimentary "situation wanted" ad. If you're available for hire and looking for a programming job (from full-time to freelance), a listing in this directory is your ticket to work. The ads are open to both 8 and 16 bit authors and are limited to 120 words or less. Be sure to give your address, phone number, and email addresses, and specify how much of a job you're after (part-time? full-time? royalty-based? etc). Send it to Situation Wanted, c/o Ariel Publishing, Box 398, Pateros, WA 98846

This month we're covering M-Z:

Eric Mueller, 2760 Roundtop Drive, Colorado Springs, CO, 80918, 719-548-8295 anytime. GEnie: [A2PRO.ERIC], CIS: 73567,1656, AO: "A2Pro Eric". Strengths include GS/OS and ProDOS 8 work, console, and modem I/O, working with hardware/firmware, desktop applications, desk accessories. Can also do tool patches, INITs, whatever. Don't call me for complex animation or sound work. Have experience working with others on programs, and on large applications. References available. Prefer 16 bit stuff always. Looking for _very_ small (less than 25 hrs/month) jobs right now.

Bryan Pietrzak, 4313 West 207th St, Matteson, Il, 60443, (708) 748-6363, or (217) 356-4351. GEnie: B.PIETRZAK1. Strengths include database design and data structures (hashing, etc) and GS/OS. Looking only for small jobs/part time work. I prefer to work in the 16-bit GS world, but can program Pascal on any system.

Lane Roath, Ideas From the Deep, 309 Oak Ridge Lane, Haughton, LA 71037. (318) 949-8264 (leave message with phone number!) or (318) 221-5134 (work). GEnie: L.Roath, Delphi: LRoath. Available for part time work, large or small for any of the Apple II line, especially the IIgs. Specializing in disk I/O graphics and application programming. Wrote Dark Castle GS, Disk Utility Package, WordWorks WP, Project Manager, DeepDOS, LaneDOS, etc. including documentation. Currently work for Softdisk G-S. Work only in Assembler.

Steve Stephenson (Synesis Systems), 2628 E. Isabella, Mesa, AZ, 85204, 602-926-8284, anytime. GEnie: [S-STEPHENSON], AOL: "Steve S816". Available for projects large or small on contract and/or royalty basis. Experienced in programming all Apple II computers (prefer IIGS), documentation writing/editing and project management. Have expertise in utilities, desk accessories, drivers, diagnostics, patching, modifying, and hardware level interfacing. Willing to maintain or customize your existing program. Work only in assembly language. Authored SQUIRT and Checkmate Technology's AppleWorks Expander, managed the ProTERM(tm) project, and co-invented MemorySaver(tm) [patent pending].

Jonah Stich, 6 Lafayette West, Princeton, NJ, 08540. (609) 683-1396, after 3:30 or on weekends. America OnLine (preferred): JonahS; GEnie: J.STICH1; InterNET: jonah@amos.ucsd.edu. Have been programming Apples for 7 years, and can speak Assembly (primary language), C, and Pascal. Currently working on the GS, extremely skilled in graphics, animation, and sound, as well as all aspects of toolbox programming. Prefer to work alone or with one or two others. Can spend about 125 hours a month on projects.

Loren W. Wright, 6 Addison Road, Nashua, NH 03062, (603)-891-2331. GEnie: [L.WRIGHT2]. Lots of experience in 6502 assembly, BASIC, C, Pascal, and PLM on a wide variety of machines: Apple II, IIgs, C64, VIC20, PET, Wang OIS. Some IIgs desktop programming. Have done several C64<>Apple program conversions. Numerous articles and regular columns in Nibble and MICRO magazines. Product reviews and beta testing. Specialties include user interface, graphics, and printer graphics. Looking for full-time work in New England and/or at-home contract work.

WE WANT YOUR BEST!

So you've written a great piece of Apple II or Apple IIGS software, but you're not sure how to turn all that hard work into hard cash. You're wary of shareware and you've been snubbed by other publishers.

Let us take a look at your work! We are the publishers of **Softdisk** and **Softdisk G-S**, monthly collections of software sold by subscription, and we're looking for top-notch Apple II and Apple IIGS software. We respond promptly, pay well, and are actually fun to work with!

Here's a short list of what will put a gleam in our eyes (and money in your pocket)! For more details, contact Jay Wilbur at (318) 221-5134.

- Teacher Utilities
  - Gradebook
  - Test Maker/Scorer
  - Attendance Keeper
  - Award Maker
- Educational Lessons
  - Geometry
  - Math
  - Physics
  - Science
- Resume Maker
- Graphical Music Maker
- Recipe Card Filer
- Magazine Indexer
- AppleWorks DB Reader
- Art Clipper DA
- Paint Program
- Cartoon Construction Kit
- Fonts
- Clip Art
- Desk Accessories

To submit your software for possible publication, send in your best to:

SOFTDISK PUBLISHING, INC.
606 Common St.
Shreveport, LA 71101
ATTN: **Apple Submissions**

This is an unpaid advertisement. Oh well.

Our Very Own Stuff

• *8/16 on Disk* •

The magazine you are now holding in your hands is but a subset of the material on the *8/16* disk. We have combed the BBS's and data services across the country to collect the best of the public domain and shareware offerings for programmers. Not only that, but we have extra articles and source code written by our staff. With DLT16 and DLT8 (**D**isplay **L**auncher **Th**ingamajigs) to guide you, you can read articles, display graphics, and even launch applications.

1 year - $69.95 6 months - $39.95 3 months - $21

• *Shem The Penman's Guide To Interactive Fiction* •

This is undoubtedly my personal favorite of all our software offerings. First of all, it is FUN. Second of all it is a very well organized, well written, and well programmed introduction to programming interactive fiction. It is, in fact, the only package of its kind I've ever seen!

Author Chet Day is a professional writer (go buy *Hacker* at your nearest book store!) and an educator who is as conerned with the *content* of your interactive fiction program as with the form. This package is fun, entertaining, and useful. It includes Applesoft, ZBasic, and Micol Advanced Basic "shells" which will drive your creations - **$39.95 (both 5.25" and 3.5" disks supplied).** P.S. The advantage to the ZBasic and Micol versions is that with the easy integration of text and graphics provided in those langauges, you can easily load a graphic and overlay text in the appropriate spots.

• *ProTools*™ • *SPECIAL PRICE THIS MONTH!*

Fast approaching its first birthday, our ProTools library for ZBasic programmers has grown into a mature and powerful product. It's bigger than ever, too. *inCider's* Joe Abernathy called it, "...the only way to go for ZBasic programmers."

ProTools includes a text based *and* a double high resolution graphics based desktop interface (pull-down menus, windows, mouse tracking, etc.) Both desktops support quick-key equivalents for menu items, too! We've added a *third* desktop package in version 2.5 of ProTools, too. This one is mouseless, meaning that it is entirely keyboard driven and therefore much more compact than its predecessors.

Mr. Ed, our "any window" text editor, will provide AppleWorks™ command compatible text editing in the screen rectangle of your choice. With no limit to edit field length, Mr.Ed is like having a word processor available as *part of your program*. Our newest version of Mr.Ed will even scroll the window if you want to support edit fields longer than your designated rectangle!

ProTools contains literally *scores* of additional functions and routines, including:

• FRAME.FN	• SMART.INPUT.FN	• SCROLL.MENU.FN
• GETMACHID	• GETKEY.FN	• SCREENDUMP80
• SAVE_SCREEN	• DIALOG	• CRYPT
• DATETIME	• BAR CHART	• LINE GRAPH
• ONLINE	• PASSWORD	• READTEXT
• SETSPEED	• VERTMENU	• PATHCK

ProTools is **$29.95 (5.25" and 3.5" disks supplied).** This is $10 off the normal price!

• *ZIndex* • (NEW! - and shipping)

If you need to write a database in ZBasic (or any other BASIC that supports multi-statement functions), *ZIndex* is the mechanism that will free you from the memory restrictions imposed by 128K Apple II's. *ZIndex* manages B+Tree indices for the key fields of your choice (it creates an index file for each key field). You can look up records in virtually any order with nearly RAM speeds, even though your data files are disk based.

ZIndex supports up to 65535 records and can perform key insertions, deletions, finds, find next, find previous, find first, find last, and find with record. The function can be used to index an existing database or a new one. It can also index unique keys or non-unique keys.

ZIndex retails for $39.95 and is shipped with both 3.5" and 5.25" disks. (Note: The current version is written specifically for ZBasic. Conversion to other BASICs may involve some translation.)

• *Micol Advanced Basic* • *SPECIAL INTRO PRICES!*

Micol Systems, Canada has produced two BASICs that should be of interest to anyone looking to empower their Apple II. Micol Advanced Basic IIe/IIc is for 128K Apples, and Micol Advanced Basic GS is for the Apple IIgs. One of the many features that recommend these two are that the GS version is upwardly compatible with IIe/IIc version. This means your 8 bit software can be quickly ported to the GS and almost immediately take advantage of the additional speed, memory, and graphics modes of the machine.

Both versions integrate graphics and text with equal ease, and both versions also provide local variables, multi-statement functions, terrific editors, multi-parameter subroutines, structured loops, and just about anything else a mature, modern language should have. The GS version has recently been extended to provide a simple interface for the creation of desktop-based programs.

MAB IIe/IIc..........$66.00	MAB GS.............$87.00

Our guarantee: Ariel Publishing guarantees your satisfaction with our entire product line (software *and* publications). If you are *ever* dissatisfied with one of our products, we will cheerfully refund the amount you paid on your request. Furthermore, we will ship the software packages to you on 30 day approval, meaning that you'll not have to pay until you've had the stuff for nearly a month. Of course, we take checks, VISA and MasterCard up front, too. Just write to: **Ariel Publishing, Box 398, Pateros, WA 98846 or call (509) 923-2249.** • We also hock some mag called *8/16.* It's 29.95 for 1 yr, $56 for 2.•

BULK RATE
U.S. POSTAGE
PAID
PATEROS, WA
PERMIT NO. 7

The Sensational Lasers

Apple IIe/IIc Compatible

SALE $375 *Includes 10 free software programs!*

New! Now Includes COPY II PLUS®

The Laser 128® features full Apple® II compatibility with an internal disk drive, serial, parallel, modem, and mouse ports. When you're ready to expand your system, there's an external drive port and expansion slot. The Laser 128 even includes 10 free software packages! Take advantage of this exceptional value today...... ***$375***

Super High Speed Option!

only **$425**

The LASER 128EX has all the features of the LASER 128, plus a triple speed processor and memory expansion to 1MB $425.00

The LASER 128EX/2 has all the features of the LASER 128EX, plus MIDI, Clock and Daisy Chain Drive Controller $465.00

DISK DRIVES

• 5.25 LASER/Apple 11c	$ 99.00
• 5.25 Apple 11e	$ 99.00
• 3.50 Apple 800K	$179.00
• 5.25 LASER Daisy Chain ... New!	$109.00
• 3.50 LASER Daisy Chain ... New!	$179.00

Save Money by Buying a Complete Package!

THE STAR a LASER 128 Computer with 12" Monochrome Monitor and the LASER 145E Printer $645.00

THE SUPERSTAR a LASER 128 Computer with 14" RGB Color Monitor and the LASER 145E Printer $825.00

ACCESSORIES

• 12" Monochrome Monitor	$ 89.00
• 14" RGB Color Monitor	$249.00
• LASER 190E Printer	$219.00
• LASER 145E Printer .... New!	$189.00
• Mouse	$ 59.00
• Joystick (3) Button	$ 29.00
• 1200/2400 Baud Modem Auto	$149.00

U.S.A. MICRO YOUR DIRECT SOURCE FOR APPLE AND IBM COMPATIBLE COMPUTERS

MasterCard VISA 2888 Bluff Street, Suite 257 • Boulder, CO 80301
Add 3% Shipping • Colorado Residents Add 3% Tax

Your satisfaction is our guarantee!

Phone Orders: 1-800-654-5426
8 – 5 Mountain Time • No Surcharge on Visa or MasterCard Orders!
Customer Service 1-800-537-8596 • In Colorado (303) 938-9089

Laser 128 is a registered trademark of Video Technology Computers, Inc. Apple, Apple IIe, Apple IIc and ImageWriter are registered trademarks of Apple Computer, Inc.