Premier Issue!
March, 1990
Vol. 1, No. 1

8/16

Welcome to our birth!

In this issue:

The Publisher's Pen, *Ross W. Lambert* 3
Filtering Out the Riff Raff, *Steve Stephenson*.............. 5
Insecticide.. 5
Getting a Jolt From Your Joystick, *David Gauger II* 10
A TransWarp GS CDEV, *Herb Hrowal* 18
Squirreling Data into Auxmem, *Ross W. Lambert* 28
The Illusion of Motion, *Steven Lepisto* 33
Them's the BRKs, *Jerry Kindall* 42
Hired Guns 46

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

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.

WARRANTY and LIMITATION of LIABILITY

Ariel Publishing, Inc. warrants that the information in 8/16 is correct and useful to somebody somewhere. Any subscriber may ask for a full refund of their last subscription payment at any time. Ariel Publishing's LIABILITY FOR ERRORS AND OMISSIONS IS LIMITED TO THIS PUBLICATION'S PURCHASE PRICE. In no case shall Ariel Publishing, Inc. Ross W. Lambert, the editorial staff, or article authors be liable for any incidental or consequential damages, nor for ANY damages in excess of the fees paid by a subscriber.

Subscribers are free to use program source code printed herein in their own compiled, stand-alone applications with no licensing application or fees required. Ariel Publishing prohibits the distribution of **source code** printed in our pages without our prior permission.

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

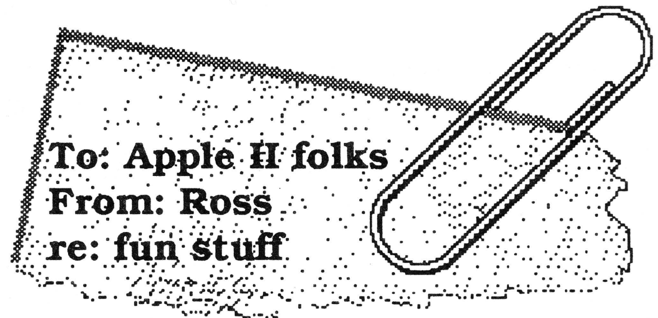
Apple, Apple II, Apple IIe, Apple IIgs, Apple IIc, Apple IIc+, AppleTalk, Apple Programmers Workshop, and Maciontosh are all registered trademarks of Apple Computers, Inc.

AppleWorks is a registered trademark of Claris, Corp.

ZBasic is a registered trademark of Zedcor, Inc.

Micol Advanced Basic is a registered trademark of Micol Sytems, Canada

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



On one hand I feel the slightest bit sacrilegious publishing a brand new Apple II technical journal before the corpse of our late , great forebear (*CALL A.P.P.L.E*) has even grown cold. On the other hand, I am pleased to have an opportunity to put my money and time where my mouth is. I have said for years that the Apple II market is still thriving for those who serve it well.

If those of us in II-dom always listened to "conventional wisdom", we would have packed it in and bought PCs or Macs at least six years ago. In spite of the Mac magazines' dire predictions, and Apple's troubles notwithstanding, I have it on good authority that 1989 was the best year *ever* for A2-Central, The ByteWorks, Applied Engineering, and of course, Ariel Publishing.

And I am far from being pessimistic about the future. The fact is that Apple's bottom line difficulties in 1989 may eventually serve to reawaken the company hierarchy. Industry analysts - both financially oriented and those of a more technical ilk - have been pillaging the company in the financial press for leaving its roots and abandoning the low end market.

What has happened, friends, is that the rest of the business world is now lending some credence to what we Apple II folks have been saying all along.

This is a good thing, though it is but a beginning.

As one computer coordinator friend of mine said, "Apple is out of their collective minds if they think we're going to supply our classrooms with Macs."

Maybe the school districts in Silicon Valley have the bucks for it, but in the rest of the USA, the Driver Education classes drive used compact cars, not Lamborghinis. Don't get me wrong, the Lamborghini is a great car and the Macintosh is a great computer. But there is most definitely a profitable place in this world for a Toyota - and the Apple II. Lest I become the target for abuse, allow me to add that in many ways I believe the Apple II is *superior* to the Macintosh, even in performance. And I am willing to bet that Toyota, Inc.

makes more moolah than Lamborghini, Inc.

"She ain't pretty, but she has a terrific personality."

Though the kiss of death to a 16 year old girl, I hope the statement above comes to describe your feelings for *8/16*. You will not find flash and splash in our pages. You won't even find any colors save black, white, and an occasional grey. Worse, you'll see page after page of text and source code listings. If there's any clip art it will be small.

So do we have some sort of twisted desire to be *ugly*? Not really. It's just that in our two years of working with Apple II programmers we have learned a few things (maybe not as quickly as our subscribers would've liked, but we did pay attention in class).

According to our surveys and interviews, you desire timely **information** more than anything else. And lots of it. We are therefore going to cram everything we possibly can into our monthly allotment of pages.

That doesn't mean that our format is encased in concrete. On the contrary, we're still brainstorming ideas for potential additions to and deletions from this magazine. And we most definitely *do* read our mail. If you have an opinion, voice it.

Where's Mike?

Mike Westerfield and The ByteWorks staff couldn't make our deadline this month. It appears we got *8/16* off the ground a little *too* fast for some! Though it pained me to go on without an Orca C article (yes, C-fans, we've been reading your letters), it was more important to be on time. Mike and Co. will be with us in the future, though, most likely on a bi-monthly basis.

The same is true of our Micol Advanced BASIC column. MAB expert Walter Torres-Hurt needed a little more time to whip his contribution into shape. Walter and the Micol gang will be with us at least bi-monthly, too.

Just because Mike and Walter are tackling C, Pascal, Orca assembler and MAB topics doesn't mean that the rest of you have to sit back and watch. We'd be delighted to consider your articles. Call or write for submission guidelines (or download them off GEnie's A2PRO RT or some other service - we're trying to get them spread all over). So you don't have to hunt around looking for our address: Ariel Publishing, Box 398 Pateros, WA 98846 Our phone: 509/923-2249. Please include a SASE.

Thanks to the advertisers!

Our advertisers in this first month of our publication have really put their money where their mouths are, too, as far as supporting the Apple II community goes. They are all companies worthy of your consideration.

For example, **So What Software's** new *Call Box* environment is much *more* than an Applesoft connection to the IIgs toolbox. Many folks have overlooked the fact it produces APW assembly source code output, too, allowing even you assembly types to design your screens, dialogs, etc. in a graphically oriented environment.

And **Night Owl Productions** has a product almost *all* of us IIgs owners will need sooner or later - an easy to install replacement battery for your IIgs. It beats taking your motherboard to a dealer (who may sit on it for a week).

USA Micro has some excellent prices on hardware as well as a proven track record. Those Laser computers are *really* a good buy, especially for your second Apple.

KAT Systems of Kansas has some great hard drive buys, as well as some of the best prices on Orca products you'll see. Say, is there some sort of Apple II commune in Overland Park, Kansas, or something?

You 8 bit programmers ought to take a serious look at **Kitchen Sink Software's** MicroDOT. This replacement for BASIC.SYSTEM saves a ton of valuable Applesoft program space and offers features BASIC.SYSTEM never dreamt about.

And of course, Ariel Publishing has a few things that may interest some of you, too! Incidentally, I think that it is just too much of a conflict of interest to do much in the way of reviewing our own packages. For that reason we'll ship them to you "On Approval". You can review them yourself.

Not so incidentally, I'd like to use this last column inch to announce an interesting experiment. As a service to Apple II programmers and companies, we will provide a *free* half page ad to any Apple II company wishing to recruit employees and announce openings.

Likewise, if you are a programmer looking for work (full time, part time, or on a royalty or contract basis), we'll offer you a free blurb to advertise yourself (c.f. p. XX). The only consideration is that you must be a subscriber, our intention being to serve our customers.

Let's hope Apple, Inc., begins to think the same way.

== Ross ==

The Merlin Maniac

Filtering Out the Riff Raff

by Steve Stephenson

As some of you may know, I've written for *Sourceror's Apprentice* in the past, and I really was honored when Ross asked me to help launch 8/16! I'm not sure why he picked me to write this column, but it might have something to with my 'bullishness' on Merlin, which I've been happily using since 1982. I'm currently doing all of my development in Merlin 16+, and find that it completely suits my needs.

In this column, I plan to show you how to 'make the magic' in assembly language and how to get more out of Merlin. I'll be focusing on 16-bit IIgs code, and my routines will usually be desktop oriented. Unless you indicate otherwise, I will assume you are interested in how to do the 'not so obvious'—in other words, if I had trouble figuring it out, someone else is probably also having trouble. This month's topic is just such a case: when I first needed to add a filter to a dialog, I couldn't find much on the subject.

When you use _ModalDialog (or any of the _Alert calls), your input is handled through filter procedures. The Dialog Manager has a built-in filter that allows the return key to act the same as clicking on the default button. It also handles cut/copy/paste operations. But what are you supposed to do when you want more? The Apple IIgs Toolbox Reference: Volume 1 devotes a whopping one page (6-25) to the subject of filter procedures, and gives no examples!

To show you how it's done, I've prepared three general purpose filters: one to handle the escape key, one to

Insecticide

So how can we be correcting errors if this is our first issue? No, we are not psychic (or psychotic) . Rather, because *8/16* is a reorganization and continuation of *The Sourceror's Apprentice*, *Znews*, and *Reboot*, we have had ample time to make boo boos. We like to correct them as soon as possible, though, so if you ever find bugs in one of our articles, please drop us a line.

• *Sourceror's Apprentice:* We discovered a mysterious insect in Jay Jennings's *Generic Start II* (September, 1989). I say "mysterious" because his source code and article were correct on the disk files he gave me, but not in the final typeset article (gremlins?). At any rate, at shutdown you need to push the *revised* StartStop record, not the original. Hence line three of the shutdown section should be: `PushLong #SSRec`

We also discovered that Steve Stephenson's tip on p.6 of the January, 1990 issue was incorrect. The long indexed addressing mode **does** roll over and cross bank boundaries. Glen Bredon himself cleared that up for us (and noted that we forgot to roll over our date on the front cover, too! Shoulda been 1990).

• *Reboot:* Subscriber Robert Lanouette wrote to say, "One comment about your Spreadsheet Mockup (October, 1989). I typed it in, then got double imaging (FILE would become FILEE when I move the cursor to BLANK, and BLANK would become BLANKK, etc.) on the top menu line whenever I moved the cursor from one menu item to another. When I changed line 740 to read H(I) = PEEK(1403) : PRINT M$(I);SPC(2) ... the double imaging disappeared." Thanks for clearingg thatt upp, Robertt.

NOTE: Back issues of *Reboot* (Applesoft programming), *The Sourceror's Apprentice* (Merlin assembly language), and *Znews* (ZBasic programming) are available for $3 per issue as long as supplies last. We've been promising an index for God-knows-how-long... soon, real soon.

gather a hex number, and one to gather a legal ProDOS name. (I've put them all together in one procedure just to keep it simple.)

To set the stage, you should imagine that your program uses two different modal dialogs, one for hex input and the other for name input. Each of these dialogs has three items: an OK button, a Cancel button, and an EditLine. As long as you're imagining, I suppose you could add a title and a prompt to the dialog, but that's not what we're here for. Note: I did not include the template to create the dialog.

The fragment of code at MainProgram should be enough to show how to call _ModalDialog with your filter. If you want your filter to be used, but would also like to get the benefit of the built-in filter, you must set bit 31 of the address of your filter when you make the call to _ModalDialog. I took the liberty of modifying the super macro to set bit 31. It is completely compatible with previous code. When you want to set bit 31, simply add a semicolon and anything for an additional parameter (it's not used for anything other than to indicate that bit 31 needs to be set).

When I call _ModalDialog, I pass it a word for the result, and the address of my filter. The Dialog Manager (among other things) rearranges the stack and calls my filter by means of a JSL. The space for the result is still there, along with pointers to the dialog's GrafPort, the event record, and the item hit. My filter routine may use the pointers any way it needs to, but it must pop all of them and put a value into the result word when it is done.

On entry, I can count on the Data Bank Register and the Direct Page being set to something that is only useful to the Dialog Manager (and it must be restored if you change it). So, I use the standard opening: save B, reset B, save D, reset D. I include a dummy section (at the label StackPic) to make it easy to access the stack/dpage variables.

On exit (see FilterDone), I restore D, B, pop all pointers off of the stack, and RTL back to the Dialog Manager. I get control again after the Dialog Manager has run it's built-in filter and possibly affected the result word. At first glance, my stack cleanup may look strange, but it is simple, easy to follow, quicker, and takes fewer bytes than the standard subtraction method.

Now, let's get down to the serious business of doing something in this filter. I start off with gathering the state of the Open Apple key and the key pressed (if any). Next, I determine what kind of event this is. If it's a key event, then I need to check for the escape key. If it is the escape key (or OpenApple-.), I simply make it look like a click in the Cancel button. If it's not an escape, I can check the key being entered into the EditLine. If it's not a key event at all, then I want the Dialog Manager to handle it. (Ignore the lines from :validate to :key for now; I'll get to them later).

Let's start with the escape key detector (see CheckEscape). If I don't find an escape, I clear the carry and return. Note that the X register, which holds False, will remain unchanged; this is to become the result unless another routine changes it. A result of False means that I didn't find anything interesting and that the Dialog Manager should handle the event. On the other hand, loading the X register with True signals the Dialog Manager that it shouldn't run through it's filter, but to just return the result.

If I find that escape is being pressed, I simulate a click in the cancel button. By protocol, the cancel button is always numbered 2 (and the default button, 1), so I make the item hit a 2 and set the result word to True. While I'm here, I think it's a nice touch to flash the button for user feedback.

The next two routines (HexCheck and NameCheck) are meant to be used with an EditLine item in the dialog. This presents a new challenge: how to filter the keys I am interested in without blocking the special editing keys used internally by the Control Manager. The way I get around the problem is to trap for the special keys first, and pass them through unaltered.

The HexCheck routine shows how to ignore (or 'swallow') any key that I don't want and how to convert a key to some other key. The way this filter works, you could pound on the Z key (or any key other than a hex digit) all day long and nothing would happen. Additionally, lowercase letters are converted as if you had entered them in uppercase. To swallow a key, set the Event record's What code to null and the Item Hit to null, then make sure to tell the Dialog Manager not to look at it either by setting the result to True. I suppose you could insert a _SysBeep here, if you're into noise.

The NameCheck routine also starts off with the edit key trap, except it makes a special case for the return key (more on that in a moment). It too allows only certain characters to pass through, while swallowing the others. I also provide a place to change the case if you desire. The last thing it does is to mark a flag that this name has changed and needs to be re-validated.

The ProDOSName routine is entirely optional. While Apple would discourage the use of such name filtering, I would maintain that there are applications where it has merit. What it does is check the name that is being gathered for meeting ProDOS naming rules and enables

the OK button only when a legal name exists. As every keystroke might result in this name changing, it needs to be checked often, but not so often as to slow everything to a crawl. I use the flag checked to prevent going through the this routine when there is no need to.

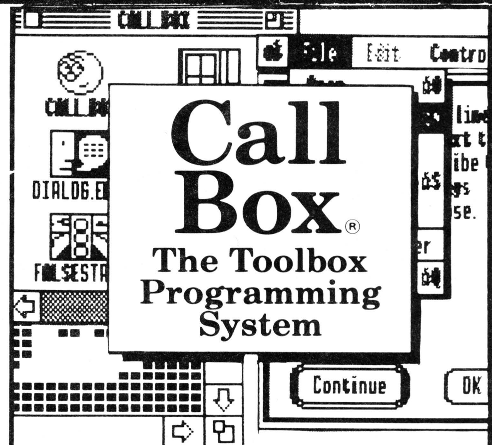
The ProDOSName routine fetches the name string (Pascal style) into a buffer. The first test is whether I have any string at all; if not, I dim the OK button. Otherwise I need to make sure the name begins with a letter. I don't need to
check further as I have already made certain no illegal characters could be entered. By the way, I control the length of the name by using the value 15 in the itemValue field when creating the EditLine.

There you have it! Custom dialog filter procedures in one easy lesson.

```
  1 *=========================================
  2 * Modal Dialog Filter Procedure          *
  3 * Copyright 1990 by Ariel Publishing     *
  4 * and Steve Stephenson                   *
  5 *=========================================
  6         use    1/tool.equates/e16.event
  7         use    1/tool.equates/e16.dialog
  8         use    1/tool.equates/e16.control
  9
 10         do     0
 11 ~ModalDialog MAC
 12
 13         DO     ]0/2
 14         PHS
 15         IF     #=]1
 16         LDA    ^]1
 17         ELSE
 18         LDA    ]1+2
 19         FIN
 20         ORA    #$8000       ; set bit 31
 21         PHA
 22         PHW    ]1
 23         ELSE               ;else, do old way
 24         P1SL   ]1
 25         FIN
 26 _ModalDialog MAC
 27         Tool   $F15
 28         <<<
 29         fin
 30
 31 NameItem =     3            ;editline item#
 32
 33 *=========================================
 34 MainProgram
 35             ...
 36
 37         ~GetNewModalDialog #Template
 38         PullLong DialogPtr
 39 :input
 40         ~ModalDialog #CustomFilter;+31
 41         pla                 ;result null?
 42         beq    :input       ; yes, go again
 43         cmp    #NameItem    ;inEdit?
 44         beq    :input       ;yes, go til btn
 45         pha                 ;no, either 1or 2
 46         ~CloseDialog DialogPtr
 47         pla
 48         cmp    #OK          ;1?
 49         beq    :done        ; yes
 50         cmp    #cancel      ;2?
 51         beq    :cancel      ; yes
 52 :done
 53             ...
 54 :cancel
 55             ...
 56
 57
 58 *=========================================
 59 CustomFilter ent
 60         phb
 61         phk                 ;push current bank
 62         plb                 ;set data bank
 63         phd
 64         tsc                 ;set stack ptr
 65         tcd                 ; to DP ptr
 66
 67 StackPic                    ;this is how the stack
 68         dum    1            ; looks right now
 69 :d      ds     2            ;orig dpage
 70 :b      ds     1            ;orig data bnk
 71 :rtl    ds     3            ;rtn addr (long)
 72 item    adrl   0            ;ptr to item hit
 73 event   adrl   0            ;ptr to event rec
 74 port    adrl   0            ;ptr to dlg port
 75 flag    dw     0            ;result to return
 76         dend
 77
 78 *-------------------------
 79
 80         ldy    #oModifiers
 81         lda    [event],y    ;get modifiers
 82         sta    KeyModifier
 83
 84         ldy    #oMessage
 85         lda    [event],y    ;get the key
 86         and    #$ff
 87         sta    theKey
 88
 89         ldx    #False       ;preset result
 90         ldy    #oWhat
 91         lda    [event],y    ;get event kind
 92         cmp    #KeyDownEvt  ;key?
 93         beq    :key         ; yes, check it
 94         cmp    #AutoKeyEvt  ;auto key?
 95         beq    :key         ; yes, check it
 96
 97 :validate                   ; not key event
 98         lda    checked ;name been checked?
 99         bne    :done        ; yes, we're done
100         jsr    ProDOSName ; no, validate it
101         ldx    #False  ;& always let Dlg Mgr
102         bra    :done        ;  handle it.
103
104 :key
105         jsr    CheckEscape ;hit escape?
106         bcs    :done        ;  yes, exit
107                             ; no, check input
```

```
108       lda    NameOrNumber
109       beq    :name
110 :number jsr  HexCheck    ;get hex digit
111       bra    :done
112 :name jsr    NameCheck;get ProDOS name chr
113 :done
114       stx    flag          ;the result
115
116 *________________________________
117
118 FilterDone
119       pld             ;restore d
120       plx             ;hold b & rtl bnk in X
121       ply             ;hold rtl addr in Y
122       pla             ;pop long (item ptr)
123       pla
124       pla             ;pop long (event ptr)
125       pla
126       pla             ;pop long (port ptr)
127       pla
128       phy             ;re-stk rtl addr
129       phx             ;re-stk rtl bnk & b
130       plb             ;restore b
131       rtl
132
133 *========================================
134
135 CheckEscape
136       lda    theKey
137       cmp    #$1b          ;escape?
138       beq    :escape       ; yes
139       cmp    #'.'          ;period?
140       bne    :none         ; no
141       lda    KeyModifier ;yes also need OA
142       bit    #AppleKey  ; got it?
143       beq    :none         ; no
144 :escape
145       lda    #2            ;# of cancel btn
146       sta    [item]        ;make it item hit
147       sta    temp       ;(also btn prt code)
148       ~GetControlDItem port;temp
149       PullLong cxHandle;cancel btn's hndl
150       ~HiliteControl temp;cxHandle;flsh on
151       ~HiliteControl #0;cxHandle ; and off
152       ldx    #True         ;,I got it,
153       sec                  ;escaping
154       bra    :done
155 :none                      ;(X & item unchanged)
156       clc                  ;not escaping
157 :done
158       rts
159
160 temp dw     0
161 cxHandle adrl  0
162
163 *=======================================
164 * keys used in an EditLine control
165
166 keys
167       dfb    $18           ;control-X
168       dfb    $19           ;control-Y
169       dfb    $06           ;control-F
170       dfb    $7f           ;delete
171       dfb    $08           ;left arrow
172       dfb    $15           ;rt arrow
173       dfb    $09           ;tab
174       dfb    $0d       ;return (leave at end)
175 keysend
176
177 *========================================
178
179 HexCheck
180       lda    theKey       ;special edit key?
181       ldx    #keysend-keys
182 :edits  shortacc
183       cmp    keys-1,x     ;chk table
184       longacc
185       beq    :pass        ; yes, leave alone
186       dex                 ;table done?
187       bne    :edits       ; no, continue
188 :normal                   ; yes, not special
189       cmp    #'0'
190       bcc    :swallow
191       cmp    #'9'+1
192       bcc    :pass        ;0-9 = good
193       cmp    #'A'
194       bcc    :swallow
195       and    #$5f         ;alpha, convert it
196       ldy    #oMessage
197       sta    [event],y    ; to uppercase
198       cmp    #'F'+1
199       bcc    :pass        ;A-F = good
200
201 :swallow                  ;gobble this key
202       lda    #0           ;make event null
203       ldy    #oWhat
204       sta    [event],y
205       sta    [item]       ;& item hit = null
206       ldx    #True   ;tell DlgMgr we got it
207       bra    :done
208 :pass
209       ldx    #False   ;let DlgMgr handle it
210 :done
211       rts
212
213 *========================================
214
215 NameCheck
216       lda    theKey
217       ldx    #keysend-keys
218       ldy    name:bad     ;if good name,
219       beq    :edits       ; <cr> is allowed
220       dex                 ;else, swallow it
221 :edits shortacc
222       cmp    keys-1,x
223       longacc
224       beq    :pass        ;special key found
225       dex
226       bne    :edits
227 :normal
228       cmp    #'.'
229       beq    :pass        ;dot = ok
230       cmp    #'0'
231       bcc    :swallow
232       cmp    #'9'+1
233       bcc    :pass        ;0-9 = ok
234       cmp    #'A'
235       bcc    :swallow
236       cmp    #'Z'+1
237       bcc    :pass        ;A-Z = ok
238       cmp    #'a'
239       bcc    :swallow
```

```
240        cmp    #'z'+1
241        bcs    :swallow   ;a-z = ok
242
243        ldy    uppercase  ;convert?
244        beq    :pass      ; no
245        and    #$5f
246        ldy    #oMessage
247        sta    [event],y  ;sneak in sub char
248        bra    :pass
249 :swallow                  ;gobble this key
250        lda    #0         ; make event null
251        ldy    #oWhat
252        sta    [event],y
253        sta    [item]     ;& item hit = null
254        ldx    #True  ;tell DlgMgr we got it
255        bra    :done
256 :pass
257        ldx    #False  ;let DlgMgr handle it
258 :done
259        stz    checked  ;mark it needing ck
260        rts
261
262 *=========================================
263
264 ProDOSName
265        ~GetIText port;#NameItem;#NewName
266        lda    NewName    ;length of string
267        and    #$ff
268        beq    :bad   ;none, skip remainder.
269
270        lda  NewName+1;1st chr mustbe alpha
271        and    #$5f
272        cmp    #'Z'+1
273        bcs    :bad
274        cmp    #'A'
275        bcs    :good
276 :bad
277        lda    #inactiveHilite ;disable item
278        bra    :hilite
279 :good
280        lda    #noHilite  ;enable item
281 :hilite
282        pha            ;(for _HiliteControl)
283        sta    name:bad   ;0=good; ff=bad
284        ~GetControlDItem port;#OK ;get hndl
285                     ;(leave handle on stk)
286        _HiliteControl    ;fix the dimming
287        inc    checked    ;mark it checked
288        rts
289
290 *=========================================
291
292 KeyModifier dw 0    ;Event modifier bits
293 theKey      dw   0  ;Event key (lo 8 bits)
294 DialogPtr adrl 0    ;ptr to the dialog
295 NameOrNumber dw 0   ;flag: which routine
296 uppercase dw    0   ;boolean: force upper
297 checked   dw    0   ;boolean: name checked
298 name:bad dw     0   ;boolean: name is bad
299 NewName   ds    16  ;name buffer
300
301 *=========================================
```

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.

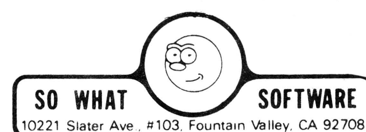
All editors output APW source code, Linkable object code or resource files to make the best match to your current development system. Everything is accessable from the CALL-BOX Editor shell that includes these editors plus File utilities, Configuration utilities, programmable application launcher and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

(714) 964-4298



The Weekend Hardware Hacker

Galvanic Skin Response: Getting a Jolt From Your Joystick

by David Gauger II

Editor: This article - and its successors since we plan more - probably seem anachronistic. I certainly know of no MacTutor subscribers tearing into their machines! But David reminded me that the Apple II has a built-in "window to the world" - and this sets it apart among all others, even today. Just try to do a similar project on a Macintosh for under $250! Apple II users have always been an outgoing, experimental lot. I think David's series is going to appeal to many of you. Even if you never actually build anything, you'll still be amazed at what the Apple II can accomplish with such relative ease.
== Ross ==

Interfacing a computer to the outside world is one of those things that falls somewhere between art and craft. An obvious solution often doesn't work as well as one that is creative or offbeat. The answers to many interfacing problems lie in rethinking them in new and different terms.

Steve Wozniak's Disk II controller card is a prime example. Prior to his design, disk controller cards used many expensive chips. To bring the cost down, Wozniak rethought the problem in different terms. Instead of relying only on hardware, he implemented many functions in software, drastically reducing the chip count. The solution he developed was so elegant that it's still in use today in all Apple II's. Even the Macintosh has its version of the controller in its "SuperWoz" chip.

In this article we will "rethink" the lowly pushbutton input in your Apple's game port and show you how to build a biofeedback monitor that interfaces to it. The usual function of the pushbutton input is to annihilate aliens in arcade games, but it's really a one-bit TTL-compatible input port capable of much more than activating phaser beams.

Pushbutton Inputs

Most Apple IIs have around 3 pushbutton inputs (the IIc and IIc+ have 2 and the GS has 4). No matter what flavor your Apple, all such inputs behave in much the same way. Like most things related to computers, we number them starting with 0. On newer Apples, the Open Apple and Solid Apple keys (Apple and Option on the IIgs) are connected to pushbutton inputs 0 and 1 in parallel to the joystick pushbuttons.

Electronically speaking, a pushbutton input simply looks at the voltage you give it to determine the button's status. If the input sees around 0 volts, the switch is considered open (not pushed). If the voltage is 5 volts or so, then it's considered closed (pushed). Each of these pushbutton (PB) inputs is electrically connected to a pin in the game input port, and, of course, to an address in memory. PB0 is at address 49249 ($C061), PB1 is at 49250 ($C062) and PB2 is at 49251 ($C063).

From the software side of things, each pushbutton is connected only to the highest bit of its address in memory. If the button is pushed, the high bit is set to one. If it's not pushed, the high bit is zero. Only the high bit is significant. Since the high bit is worth 128 in the binary number system it is easy to tell when the button is pushed. All you need to do is check to see if the value at the pushbutton's address is greater than 127. If so, the button is pressed. In assembly language, the check can be most easily performed with the BMI or BPL instructions.

Most programs use only the minimum information that this one-bit input can provide. In an arcade game, when you push the button, the phaser fires; when you release it, the phaser stops. This is fine, but it leaves much of the port's capability untapped.

Let's rethink the use of this port by shifting our focus from a mere on-off reading to reading how often it is pushed in a given period of time. In other words, the rate of button pushing is the significant thing. This opens up many possibilities since the port then becomes capable of receiving analog (continuously variable) data, not just binary (strictly on or off) data. We can receive analog data through this port at relatively high speeds, since an input port can be checked for activity very rapidly in machine language (over 140,000 times a second in a 1 mhz Apple).

Biofeedback and Button Pushing

Biofeedback has been around for a while but surpris-

ingly few people have ever tried it. Biofeedback measures a biological function that correlates to your tension level, then feeds that information back to you in real time. Immediate feedback enables you to identify the things that make you tense, and discover those techniques most effective in promoting relaxation.

One biological function that varies with tension level is the electrical resistance of your skin. The more tense you are, the more you sweat; and the more you sweat, the lower your skin resistance. This skin characteristic is known as galvanic skin response, or GSR for short. Because people often get tense and sweat when not telling the truth, GSR is one of several measurements made by a lie detector.

Your health, level of physical activity, the temperature of your hands, and even the weather can all have a profound effect on your GSR. Your GSR is constantly changing. Because it's not stable, GSR is usually interpreted as a relative phenomena. The most common GSR technique is to first establish a normal or baseline GSR, then note changes or departures from the norm.

Your Apple II computer is well-suited to making the GSR measurements for a biofeedback system. Not only can the computer collect lots of data, but it can display the data graphically, manipulate it, scale it, compare it to previous measurements, and store it on disk for later use, to name just a few ideas. The pushbutton input is up to the task, but it'll need the help of some simple hardware and a small machine language driver.

Construction

To get your biofeedback monitoring system up and running, you'll need to build a device that converts your GSR into a signal usable by the pushbutton input. Two schematic diagrams (Fig. 1 and 2) are provided, although the circuit is identical in each case. The only difference is the connector used to plug the device into your Apple.

Build your biofeedback monitor in a small plastic experimenter's box such as those sold by Radio Shack. Use the parts list in Fig. 3 if your computer has a DB-9 joystick connector, and the list in Fig. 4 if you have a 16-pin DIP connector. GSR measurements are collected with two parallel 5/8 inch by 2 1/2 inch strips of common aluminum foil glued to the top of the plastic box. Glue the dull side down with rubber cement and separate the strips by about 3/8 inch. (See Fig. 5).

Drill two holes through the plastic, 1/4 inch from the same end of each strip. A machine screw's head will contact the foil and allow connection between the foil and the rest of the circuitry. You can connect the wires to the screw with a lug, or by wrapping the wire around the screw before tightening the nut.

It is probably best to mount the components on a small 1 inch square circuit board cut from a larger one. Once it's working, the circuit board can be stuck to the bottom of the box with double sided tape or silicon sealer. The rest of the construction is not critical. Use common construction techniques, insulating all bare wires with black electrician's tape or shrink tubing, and checking for shorts caused by things like stray bits of solder.

Double check your wiring, especially the connector going to your Apple. It's easy to confuse the pins. We're only dealing with about 5 volts or so, so danger from voltage is not a problem, but wrong connections to the gameport could easily damage your Apple.

The Software

Next, type in the Applesoft listing and save it on disk with the name "BIOFEEDBACK". If you have an assembler, type in the assembler source code and assemble it using the instructions appropriate for your assembler. Otherwise, enter the object code from the monitor and save it with the command BSAVE CYCLETIMER.OBJ,A$303,L$3A.

These programs work under DOS 3.3 or ProDOS, but both files must be on the same disk (and in the same directory if you're using ProDOS.)

Using the Biofeedback System

To use the complete system, plug your Biofeedback monitor into the correct connector on your Apple, turn the switch on the biofeedback sensor to the on position, and run the program BIOFEEDBACK. Make sure you are in 40 column mode with the checkerboard cursor active before you start.

First, the program needs to calibrate itself to your baseline GSR. Place two fingers of one hand on the aluminum strips. The biofeedback hardware is sensitive to movement and pressure, so position your arm and hand to prevent any pressure variation of your fingers on the aluminum strips. Your arm, hand, and fingers should be as relaxed as possible. When you are ready to begin calibration, hit any key. Calibration takes a few seconds, and then the graph of your GSR will start appearing on the screen along with many session statistics.

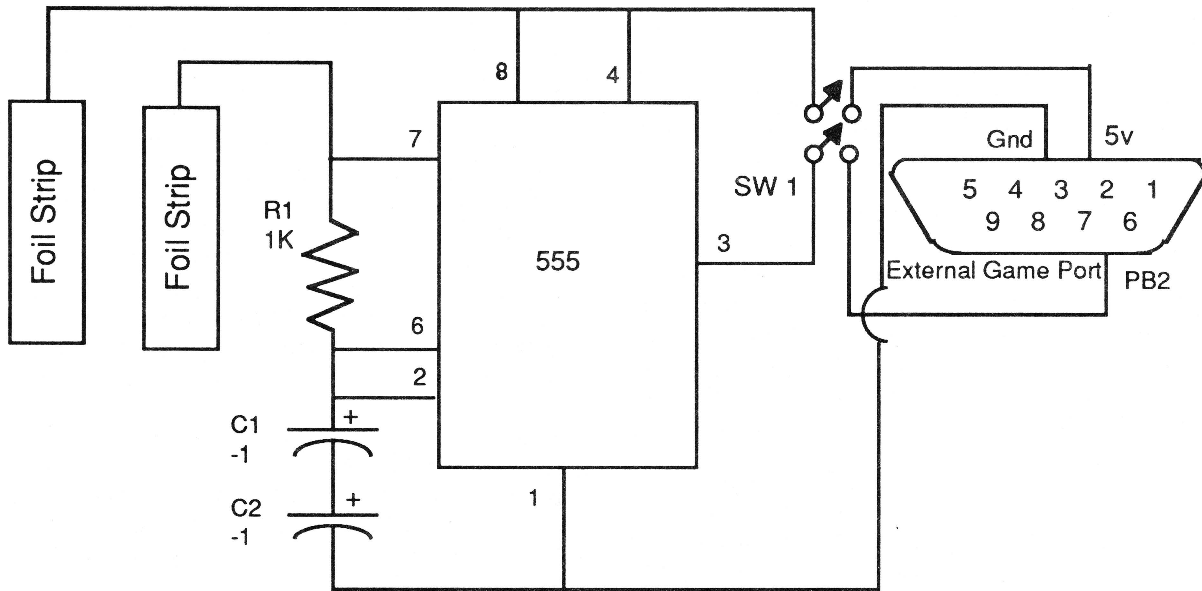


Figure 1 - Schematic Diagram for DB-9 Connectors (Apple IIc, IIc+, IIe, IIgs)

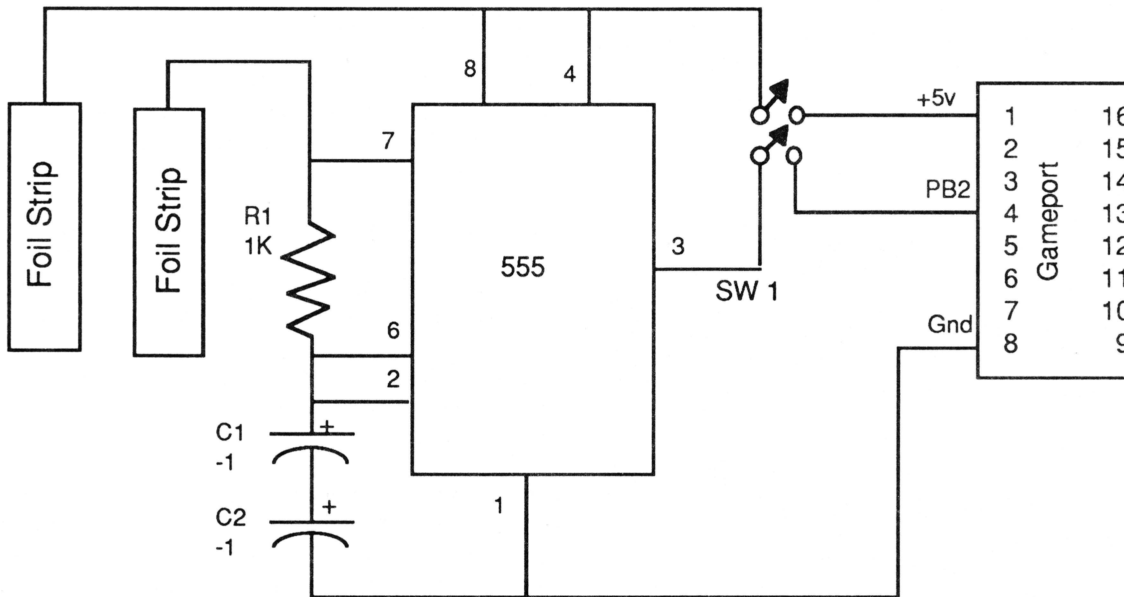


Figure 2 - Schematic Diagram for 16 pin DIP Connectors (Apple II, II+, IIe, IIgs)

Figure 3: Parts List for DB-9 Version

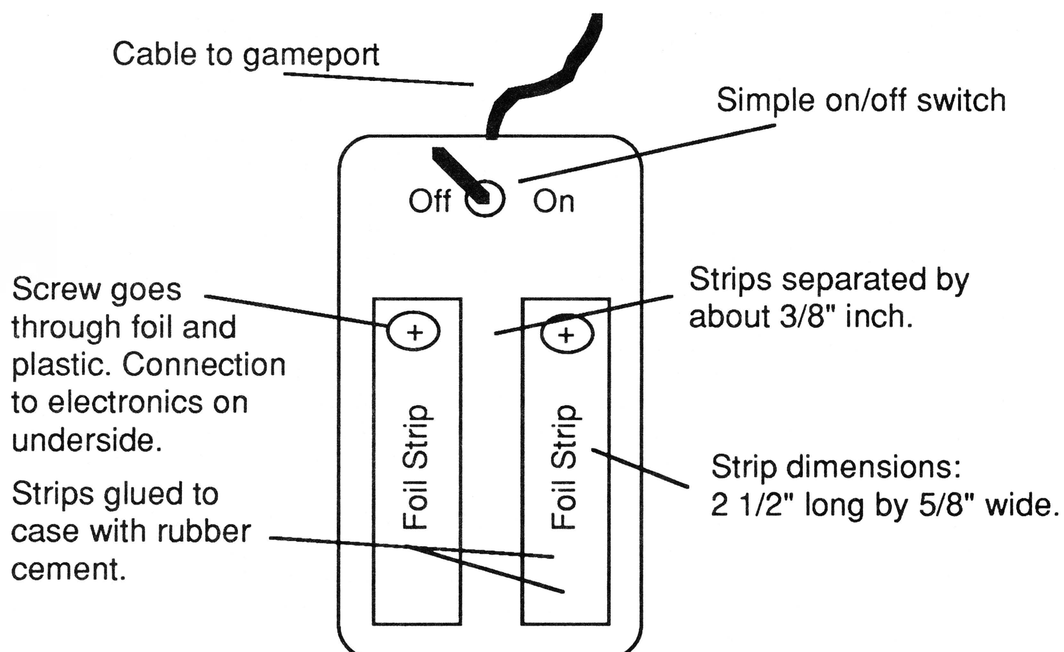
Quantity	Description	Radio Shack Part #	Cost
1	555 Timer	276-1723	$1.19
2	.1 Mfd Tantalum Capacitors	272-1432B	.59 ea
1	1K ohm Resistor	271-023	.19
1	DPDT Toggle Switch	275-626	2.59
3 ft	Shielded 2 Conductor Wire	278-1276	3.29
1	Experimenter's Box	270-220	1.79
1	DB-9 Male Connector	276-1537	.99
		Total	$11.22

Figure 4: Parts List for 16 Pin DIP Version

Quantity	Description	Radio Shack Part #	Cost
1	555 Timer	276-1723	$1.19
2	.1 Mfd Tantalum Capacitors	272-1432B	.59 ea
1	1K ohm Resistor	271-023	.19
1	DPDT Toggle Switch	275-626	2.59
3 ft	Shielded 2 Conductor Wire	278-1276	3.29
1	Experimenter's Box	270-220	1.79
1	16 Pin DIP Header	*N/A	.65
		Total	$10.88

* The 16 pin DIP header is not made by Radio Shack anymore. Instead, it is part # 16 HP available from Jameco Electronics, 1355 Shoreway Road, Belmont, CA 94002 (415) 592-8097

Figure 5 - GSR Measuring Device



As the program runs, it measures your GSR, plots the results on the graph portion of the screen, and updates the statistics at the bottom. This process continues until the graph reaches the right side of the screen at which point it wraps around and begins plotting over the previous data on the left side of the screen. When the graph wraps around, the screen average (SCR AVG) starts over; the rest of the statistical data (HIGHEST, LOWEST, etc.) is valid and carries over into the new screen.

The top of the screen represents high tension while the bottom represents lower tension. Remember that a high GSR reading means that your skin resistance was high because you weren't sweating, so you must be relaxed. Conversely, a low GSR means that you're tense. To represent them on the graph correctly, low readings are plotted at the top of the screen and high readings at the bottom, an important fact to remember when evaluating the statistical data at the bottom of the screen.

Because GSR readings can vary so widely, the program provides two sensitivity levels. High sensitivity is the most accurate, but the plot is a jagged line that often goes off the screen. You can remedy this situation by recalibrating to bring the plot back to the center of the screen, or by switching to low sensitivity. Low sensitivity attempts to scale the data so that all of it fits on the screen, but the changes in the GSR graph are less obvious. Try them both and use the one that is most helpful to you at the time.

Changing the sensitivity level does not change the data collected, only the way the data are represented on the graph. Changing the sensitivity level or recalibrating causes the program to regraph old data in terms of the new setting to allow new readings to be compared with the older data on the graph.

The Statistics

The statistics at the bottom of the screen are a numerical summary of what is represented on the graph. Remember that a high number represents high skin resistance and is associated with a low level of tension.

Current: the reading currently being plotted.
Base: the base GSR measurement currently used to plot the graph. This number changes each time you recalibrate.
Scr Avg: average of the newest data plotted. This includes all data to the left of the moving cursor.
Sens: the current sensitivity setting.
Reading #: the number of the current reading. All readings are numbered sequentially since the beginning of the session.
Highest: the highest reading (lowest tension level) detected since the beginning of the session.
Lowest: the lowest reading (highest tension level) detected since the beginning of the session.
Span: The difference between the highest and lowest readings.

The Options

Several options are available while the program is running. Hitting the following keys while the program is graphing your GSR will have the listed effect:

Key	Effect
C: Calibrate	If your readings go off the screen recalibration will put them back i the center of the Hires graph. Da are regraphed in terms of the new calibration.
S: Sensitivity	Sensitvity is retoggled and data are regraphed.
P: Pause	Hit any key while in the Pause mod to resume.
?:	Help Displays these choices at bottom of screen. Hit any key to resume.
Q: Quit	Quit the session.

Theory of Operation

Getting your skin to talk to your Apple (i.e. measuring your GSR) presents a bit of a problem. The first solution might appear to use the game controller inputs since they measure resistance directly, but the values of a typical GSR lay that idea to rest. For example, my GSR has varied from about 20K ohms to almost 500K ohms. Since the game controllers are capable of measuring from 0 to only 150K ohms, many GSR measurements will be beyond the reach of the game controller input.

A better solution is to use the technique described earlier: employing a pushbutton input, transfer the data by varying the rate of "button pushes". A low GSR reading yields closely-spaced "pushes"; higher GSR readings come farther apart. The electronics are acutally rather simple, so let's delve into the technical side of things for a moment. If electronics don't interest you, just skim this next part; you don't have to understand every connection to use your biofeedback monitor.

The Hardware Interface

Normally, the PB inputs are at ground level: when they aren't pushed, the voltage on their respective pins in the game port is 0. When you want to "push the button", you connect the PB input to +5 volts. All we have to do to register a button push is apply +5 volts to the correct pin for the length of time we want the button pushed. There doesn't have to be a physical switch present since the only important thing is how much voltage the PB port sees.

There are many ways to swing the voltage on the PB input pin between 0 and +5 volts. Joysticks and paddles use a switch to connect the PB port to the 5 volt supply found in the game connector itself. The GSR monitor uses a different technique: a type of oscillator called an astable multivibrator whose output swings between these two voltages at a regular rate of speed. A chip called a 555 timer does this job very well.

The 555 circuit used to generate this swinging voltage includes a resistor and a capacitor, which, together, determine the oscillation frequency. The number of times per second the 555 timer swings between 0 and 5 volts is directly proportional to the values of these two components. If you build the circuit with a fixed capacitor and a fixed resistor, you'll always get the same rate of oscillation. If you make either component variable, you can vary the output frequency.

Suppose we use a fixed capacitor, but substitute your changing GSR as the resistor portion of the circuit (after all, GSR is a resistance). Now when your GSR shifts, the circuit sees a changing resistance, and the oscillation frequency varies by a proportional amount.

Since the output of the 555 swings between 0 and 5 volts, its output can be directly connected to a pushbutton input in the game port and the game port will be fooled into thinking a button is being pushed. So when your skin resistance decreases due to sweat, the PB port will see an increase in the number of button pushes per second. Should your skin resistance rise, the PB port will detect fewer pushes per second. Measuring your GSR becomes a simple matter of measuring the elapsed time between pushes. (See Fig. 6)

The Machine Language Driver

The main job of the software portion of our interface (the driver) is to measure the time between button pushes. Most time-dependent tasks are best dealt with in machine language since its faster execution speed makes precision easier to achieve. CYCLETIMER is a machine language subroutine called from the main Applesoft program that does nothing but measure the length of time between pushes.

Looking at this from the game port's point of view, what CYCLETIMER will see is a bunch of shifts between 0 and 5 volts. To make any sense out of this we need to think of these voltage changes as cycles. One complete cycle consists of a period of time where the voltage is 5 volts followed by a period of time where it is 0 volts.

CYCLETIMER's job is to measure (time) the length of one complete cycle. To be consistent, the routine must start measuring at the same point in each cycle. CYCLETIMER accomplishes this by triggering only on the rising edge. In other words, it is sensitive only to the transition from 0 to 5 volts.

One cycle can be defined as the time between sequential rising edges, so CYCLETIMER just finds the first rising edge and begins counting the time period, counts through the 5 volt portion of the cycle, keeps counting when the voltage returns to 0 volts, and finally stops counting the instant the voltage begins its rise to 5 volts again.

CYCLETIMER measures (or counts) the length of time between each button push by going around in a loop, incrementing a counter each pass. When the second rising edge is detected, we know the cycle is over, so the routine exits, and the the count then reflects the length of the cycle. The larger the count, the longer the period of the cycle.

The counter in CYCLETIMER is actually a 3 byte number maintained in locations 768, 769 and 770. Location 768 ($300) contains the number of ones, while 769 and 770 hold the number of groups of 256 and 65536, respectively.

The Applesoft Program - How It Works

BIOFEEDBACK is not a long program since its main function is just to display data the hardware and software interfaces deliver to it. Most of the hard work is done by the hardware and CYCLETIMER. The subroutine at 990 is the initialization routine. The code at 680 performs calibration by taking 100 readings and totaling them; the total (TTL) is used in determining how to scale the data coming in. The routine at 590 actually reads the data collected by the interface. Line 630 PEEKs the data passed in locations 768-770 into a numeric variable.

Each point plotted on the screen is really an average of 10 readings to smooth out the potentially jagged-looking graph. At line 420 is a subroutine that updates all

the numerical data at the bottom of the screen.

The main program loop is in lines 150-280. This loop calls most of the other subroutines and scales the data. The sensitivity levels are represented by the variable SENS. If SENS is one, the average of your calibration data is put at the center line on the screen, doing no scaling at all (a change of one in the data will be reflected as a change of one point on the graph). If SENS is zero, the program scales the data with the attendant reduction in sensitivity.

Modifications and Improvements

If you have an unaccelerated Apple II+, IIe or IIc, you might want the program to plot readings a little faster than it does. Recall that each point plotted on the screen is actually ten GSR readings averaged together. If we lower the number of readings take in line 610 to 5 (FOR Y = 1 TO 5) and correct line 660 to agree with it, (GSR = INT (Y/5)), the program will plot readings roughly twice as fast.

The BIOFEEDBACK program presented here could be considered unfinished. Many enhancements could be added. For example, session data could be saved to disk, audio feedback could be added, and other types of statistics can be collected and displayed. The whole system could be rewritten to support the mouse and use a windowing interface. Or... maybe the we need to rethink the project in completely new terms!

Figure 6 - Cycle Time vs. GSR

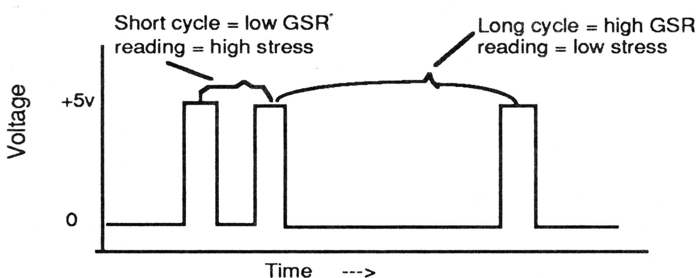
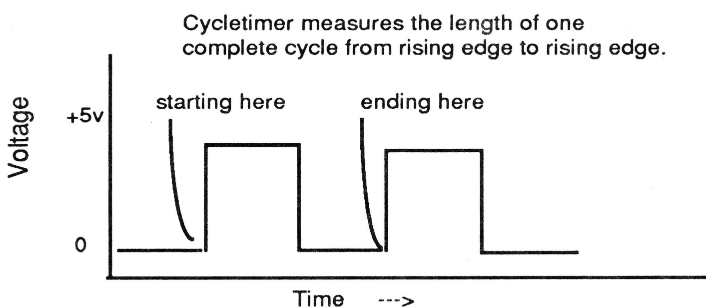


Figure 6 - Cycle Time vs. GSR



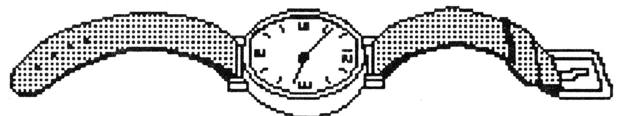
Editor: A note about our Applesoft listings - in an effort to converve space, most of our program listings will be single column. This means that many lines will wrap, particularly in Applesoft programs. We are experimenting with different layouts, but from our jaundiced viewpoint, it appears that indenting wrapped lines is one of the best options. Let us know what you think. == Ross =

BIOFEEDBACK Program Listing

```
100  REM  BIOFEEDBACK
110  REM  1-11-90
120  REM  (C) 1990
130  REM  BY DAVID GAUGER II
140  GOSUB 1000: GOSUB 690: GOSUB 960:GOSUB 890
150  REM  MAIN LOOP
160  TALLY = 0
170  CNTR = 1:TTL = GSR
180  KEY =  PEEK (KBD): IF KEY > 127 THEN 300
190  GOSUB 600
200  ARRAY(CNTR) = GSR
210  IF SENS$ ="HIGH" THEN FUDGE = BASE-80:
         DOTLOC = GSR - FUDGE
220  IF SENS$ = "LOW" THEN FUDGE = 80 /
         BASE:DOTLOC = GSR * FUDGE
230  IF DOTLOC < 0 THEN DOTLOC = 0
240  IF DOTLOC > 159 THEN DOTLOC = 159
250  HCOLOR= 0: HPLOT CNTR,0 TO CNTR,158:
         HCOLOR= 3: HPLOT CNTR,DOTLOC
260  HCOLOR= 3: HPLOT CNTR + 1,0 TO CNTR +
         1,158
270  CNTR = CNTR + 1: IF CNTR = 279 THEN
         HCOLOR= 0: HPLOT CNTR,0 TO CNTR,158
         :GOTO 170
280  GOSUB 430: GOTO 180
290  REM  KEYBOARD
300  POKE STB,0:KEY=KEY-128:KEY$ =  CHR$ (KEY)
310  IF KEY$ = "C" THEN GOSUB 690: GOSUB 960:
         GOSUB 790: GOSUB 890: GOTO 180
320  IF KEY$ = "S" THEN  IF SENS$ = "HIGH" THEN
         SENS$ = "LOW ": VTAB 24: HTAB 11PRINT
         SENS$;:GOSUB 960: GOSUB 790: GOTO 180
330  IF KEY$ = "S" THEN  IF SENS$ ="LOW " THEN
         SENS$ = "HIGH": VTAB 24: HTAB 11:
         PRINT SENS$;:GOSUB 960: GOSUB 790:
         GOTO 180
340 IF KEY$ = "P" THEN  HOME : VTAB 22:
         HTAB 1:PRINT"IN PAUSE MODE: HIT ANY
         KEY TO CONTINUE.";: GET A$: HOME :
         GOSUB 890: GOTO 180
350  IF KEY$ = "/" THEN  GOSUB 380: HOME :
         GOSUB 890: GOTO 180
360  IF KEY$ = "Q" THEN  HOME : VTAB 22: HTAB
         5:PRINT "SURE YOU WANT TO QUIT?(Y/N)";
         :GET A$:IF A$ = "Y" THEN  TEXT : HOME
         : END
370  GOSUB 890: GOTO 180
380  HOME : VTAB 21: HTAB 1
390  HTAB 5: PRINT "S = TOGGLE SENS.  C =
         CALIBRATE"
400  HTAB 5: PRINT "P = PAUSE          Q = QUIT"
410  VTAB 24: HTAB 5: PRINT "HIT ANY KEY TO
         CONTINUE...";: GET A$: RETURN
```

```
420  REM  UPDATE STATISTICS
430  VTAB 21: HTAB 11: PRINT "      ";
440  VTAB 21: HTAB 11: PRINT GSR
450  TALLY = TALLY + 1:TTL = TTL + GSR
460  VTAB 21: HTAB 31: PRINT TALLY
470  VTAB 22: HTAB 11: PRINT "      "
480  VTAB 22: HTAB 11: PRINT BASE
490  IF GSR > HIGHEST THEN HIGHEST = GSR
500  VTAB 22: HTAB 31: PRINT HIGHEST
510  VTAB 23: HTAB 11: PRINT "      "
520  VTAB 23: HTAB 11: PRINT  INT (TTL / CNTR)
530  VTAB 23: HTAB 31: PRINT "      ";
540  IF GSR < LOWEST THEN LOWEST = GSR
550  VTAB 23: HTAB 31: PRINT LOWEST
560  VTAB 24: HTAB 11: PRINT SENS$;
570  VTAB 24: HTAB 31: PRINT HIGHEST - LOWEST;
580  RETURN
590  REM  GET A GSR READING
600 Y = 0
610  FOR X = 1 TO 10
620  CALL 771
630 GSR =  PEEK (770) * 256 ^ 2 +  PEEK (769) *
        256 +  PEEK (768)
640 Y = Y + GSR
650  NEXT X
660  GSR =  INT (Y / 10)
670  RETURN
680  REM  CALIBRATE
690  HOME :TL = 0: VTAB 2: HTAB 14: INVERSE :
        PRINT "BIOFEEDBACK": NORMAL :VTAB 5:
        HTAB 11:PRINT "BY DAVID GAUGER II"
700  VTAB 10: HTAB 4: PRINT "(BE SURE CAPS LOCK
        KEY IS DOWN.)"
710  VTAB 22: HTAB 1: PRINT "HIT ANY KEY TO
        BEGIN CALIBRATION...";: GET A$
720  HOME : VTAB 23: HTAB 15: FLASH : PRINT
        "CALIBRATING": NORMAL
730  FOR Z = 1 TO 10
740  GOSUB 600:TL = TL + GSR
750  NEXT Z
760  BASE =  INT (TL / 10)
770  RETURN
780  REM  REDRAW DATA
790  X = 0
800  X = X + 1
810  IF X > 279 THEN  RETURN
820  IF ARRAY(X) = 0 THEN  RETURN
830  IF SENS$ = "HIGH" THEN DOTLOC = ARRAY(X) -
        (BASE - 80)
840  IF SENS$ = "LOW " THEN DOTLOC = ARRAY(X) *
        (80 / BASE)
850  IF DOTLOC > 158 THEN DOTLOC = 158
860  IF DOTLOC < 0 THEN DOTLOC = 0
870  HPLOT X,DOTLOC: GOTO 800
880  REM  INIT STATISTICS DISPLAY
890  HOME
900  VTAB 21: HTAB 1: PRINT "CURRENT = ";: HTAB
        19: PRINT "READING # = ";
910  VTAB 22: HTAB 1: PRINT "BASE    = ";: HTAB
        19: PRINT "HIGHEST   = ";
920  VTAB 23: HTAB 1: PRINT "SCR AVG = ";: HTAB
        19: PRINT "LOWEST    = ";
930  VTAB 24: HTAB 1: PRINT "SENS    = ";: HTAB
        19: PRINT "SPAN      = ";
940  RETURN
950  REM  SETUP HIRES SCREEN
960  HGR : HCOLOR= 3
970  HPLOT 0,0 TO 0,159 TO 279,159
980  RETURN
990  REM  INIT
1000 SENS$ = "LOW ": DIM ARRAY(279)
1010 LOWEST = 10000:HIGHEST = 0:KBD = 49152:STB
        = 49168
1020  PRINT  CHR$ (4)"BLOAD CYCLETIMER.OBJ"
1030  RETURN
```

Cycletimer Listing

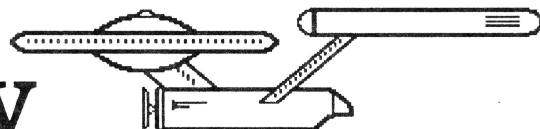


```
              1     ********************************
              2     *
              3     * CYCLETIMER
              4     *
              5     * by David Gauger II
              6     *
              7     * (C) 1989
              8     *
              9     * Merlin 8 Assembler
              10    *
              11    ********************************
              12
              13              org     $303
              14
              15    low       =       $300      ;ones passed to BASIC here
              16    med       =       $301      ;groups of 256 passed here
              17    high      =       $302      ;groups of 65536 passed here
              18    bttn      =       $C063     ;address of pushbutton #3
              19
0303: A9 00   20    start     lda     #00       ;init counters
```

```
0305: 8D 00 03   21                 sta    low
0308: 8D 01 03   22                 sta    med
030B: 8D 02 03   23                 sta    high
030E: 2C 63 C0   24      wavehi     bit    bttn       ;positive 1/2 of cycle?
0311: 30 FB      25                 bmi    wavehi     ;yes: loop until negative
0313: 2C 63 C0   26      wavelo     bit    bttn       ;negative 1/2 of cycle?
0316: 10 FB      27                 bpl    wavelo     ;yes: wait for 1st rising edge
0318: 20 29 03   28      posjsr     update            ;update counters (positive half)
031B: 2C 63 C0   29                 bit    bttn       ;still positive 1/2 of cycle?
031E: 30 F8      30                 bmi    pos        ;yes - keep counting
0320: 20 29 03   31      neg        jsr    update     ;no: neg now, but keep counting
0323: 2C 63 C0   32                 bit    bttn       ;still negative 1/2 of cycle?
0326: 10 F8      33                 bpl    neg        ;yes - keep counting
0328: 60 34              dun        rts               ;no: found 2nd rising edge.
         35
0329: EE 00 03   36      update     inc    low        ;update counters here
032C: F0 03      37                 beq    next1      ;if rolled over, inc 256'S (med)
032E: 4C 3C 03   38                 jmp    done       ;otherwise just rts
0331: EE 01 03   39      next1      inc    med        ;next 256
0334: F0 03      40                 beq    next2      ;if rolled over, inc 65536'S
0336: 4C 3C 03   41                 jmp    done       ;otherwise rts
0339: EE 02 03   42      next2      inc    high       ;next 65536
033C: 60 43              done       rts               ;don't worry for 65536 rollover
```

I'm givin' her all she's got, Captain!

A TransWarp GS CDEV



by Herb Hrowal

With the advent of System Disk 5.0, a new method of modifying the settings of the Apple IIgs is available, through the Control Panel NDA. Unlike the text-based control panel (accessed from the CDA menu), the NDA is much more flexible in that it is easily expanded with additional CDEVs (Control panel DEVices).

Every time the Control Panel NDA is selected from the apple menu of a desktop application, it scans the */SYSTEM/CDEVS folder of the boot device searching for all files with a type of $C7/$0000. Each CDEV found is recorded by the NDA in a separate file (*/SYSTEM/CDEVS/CDEV.DATA). The icon and title for each CDEV is displayed in the NDA window, and can be clicked on to activate the corresponding CDEV.

The purpose of this article is to show how you can create your own CDEV with very little effort. CDEVs are not restricted to modifying the system settings—some other uses for them might be a screen saver, or to control a piece of hardware in your computer. In this article, I will show how to write a CDEV to control the popular TransWarp GS accelerator card.

Let's take a look at the Rez source file (listing one) first.

Each CDEV is required to include three resource types, in a predetermined order, with an ID of 1. The first resource in the file must be a standard icon resource, type $8001. This is the icon that will be displayed in the NDA window. If the CDEV has an initialization routine (discussed later), the icon will also be displayed on the bottom left corner of the screen at boot time. One thing to be aware of with the icon is that it must have a height of 20 and a width of 28 in order to be displayed properly during boot up. The reason for this is that the GS/OS boot code does not use the size information supplied with your icon, but is hard coded for an icon that is 20 X 28. If you don't have an initialization routine, the icon can be any legal size.

Here's what the icon resource looks like in Rez code:

```
/* resource type $8001, ID = 1 */
```

```
resource rIcon (1) {
(icon flags),
(icon height),
(icon width),
(icon data),
(mask data),
};
```

The second required resource is the actual program code that does the work for the CDEV. The CDEV code resource (type $8018) is simply your application, turned into a CODE resource and inserted into the CDEV resource fork. This resource is then loaded and called when the CDEV's icon is selected in the NDA. The maximum size of the CDEV code resource is 64K, which is more than enough for most applications.

The command (in the Rez source code) to do this is:

```
read rCDEVCode (1,convert) "tw.code";
```

The third and final required resource is the CDEV flags resource (type $8019). This resource tells the Control Panel NDA what types of events the CDEV can handle and also contains the data rectangle, the title, the author, and the version of the CDEV. Again, the Rez code you need to have looks like this:

```
resource rCDEVFlags (1) {
        flags,
        enabled,
        version,
        machine,
        reserved,
        data rectangle,
        title,
        author,
        version name,
};
```

Let's start at the bottom of these flags and work our way up.

"Version Name" is an 8 character string that is displayed in the upper right hand corner of the Help/About box that is created every time the user clicks on the Help button.

"Author" is a 32 character string containing the name of the author of the CDEV. This is also displayed in the Help/About box.

"Title" is a 15 character string containing the name of the CDEV. This is displayed with the icon in the NDA's scrollable window and in the Help/About box.

"Data Rectangle" is four words (a standard RECT) containing the size of the window the CDEV needs to work in. The top left corner must be 0,0.

"Machine" is a one byte field containing the minimum ROM version needed for the CDEV to run. For example, this could be useful when writing a CDEV that needs a ROM 03 GS to run.

"Version" is a one byte field containing the version number of the CDEV.

"Enabled" is a one byte field used to determined whether the CDEV can be activated or not. When this byte is zero, the CDEV can't be used.

"Flags" is a one word field containing flags used to enable the type of events the CDEV wants to receive. The flag definitions are:

Bit	Name	Enabled Event
15 thru 11		Reserved, must be zero
10	wantRun	RunCDEV
9	wantHit	HitCDEV
8	wantRect	RectCDEV
7	wantAbout	AboutCDEV
6	wantCreate	CreateCDEV
5	wantEvent	EventCDEV
4	wantClose	CloseCDEV
3	wantInit	InitCDEV
2	Reserved, must be zero	
1	wantBoot	BootCDEV
0	Reserved, must be zero	

Following the 3 required resources, any number of additional resources may be used in any order you like. (The rest of the Rez code, in this case, is templates for the help/about screen, the menus, and the controls used in the TWGS CDEV.) All controls in the window must be 'super controls' created with _NewControl2, since the Control Panel NDA uses _TaskMasterDA to track them. For more information on the format of CDEVs, refer to Apple II File Type Note $C7 (September 1989).

Now that we have the Rez portion out of the way, we can take a look at the actual code to control the CDEV. I wrote this code (listing two) in assembly language with Merlin 16+, but have also written CDEVs in C, and they can be done in Pascal as well.

Whenever an event occurs in your window, the CDEV Code resource is called "tool-style" by the Control Panel NDA. This means it pushes parameters on the stack and calls the CDEV similar to the way you make a toolbox call. It is the CDEV's responsibility to extract the

parameters from the stack and fix the stack pointer before it returns control back to the NDA. The parameters passed are:

```
Long     Result
Word     Message
Long     Data1
Long     Data2
3 bytes RTL Address
```

There are 11 possible values for Message. Currently, message three is reserved for future use as a shutdown message. I also tried using MachineCDEV (explained after the table) to enable or disable the CDEV depending on whether or not a TransWarp GS was installed, but it didn't seem to work, so I abandoned the idea.

Here's a table of each message, and the associated data passed with them:

Message	Data1	Data2	Result
1 - MachineCDEV	undef	undef	0 = inactive
2 - BootCDEV	undef	undef	undef
3 - Reserved	undef	undef	undef
4 - InitCDEV	wnd ptr	undef	undef
5 - CloseCDEV	undef	undef	undef
6 - EventsCDEV	evntrec ptr	undef	undef
7 - CreateCDEV	wnd ptr	undef	undef
8 - AboutCDEV	wnd ptr	undef	undef
9 - RectCDEV	rect ptr	undef	undef
10 - HitCDEV	ctrl hndl	ctrl ID	undef
11 - RunCDEV	undef	undef	undef

MachineCDEV is called when the NDA is activated. This is an ideal place to determine if the CDEV should be displayed or not. If zero is returned in Result, the icon is not displayed. Currently it is not possible to enable MachineCDEV, but should be available in the future.

BootCDEV is called at boot time and should contain the code to initialize the peripheral you are controlling (if necessary). The icon for the CDEV will be displayed on the bottom right of the screen for the duration of the initialization routine.

InitCDEV is called after CreateCDEV, but before the controls are displayed. This is where you should initialize all the controls in the window.

CloseCDEV is called when the NDA window is closed or another CDEV is selected. Housekeeping should be taken care of in this routine.

EventsCDEV is called before the event record is passed to TaskMasterDA, allowing you to intercept or even change the event record before processing (if desired).

CreateCDEV is called when CDEV is selected. This is where the controls should be created. Initialization of the controls should be done in InitCDEV.

AboutCDEV is called when the user clicks on the Help button on the bottom of the NDA window.

RectCDEV is called before the window is displayed and is used to change the
size of your display window if it needs resizing.

HitCDEV is called every time one of your controls in the window is clicked on or "hit".

RunCDEV is called 60 times per second and is an ideal place to update a clock display, poll disk devices for a certain volume, or just to make sure the controls don't need updating.

The first thing we need to do when the code is called is set the bank register (since we don't know where we'll be in memory), get the parameters off the stack, fix the stack pointer, and call the proper routine. When the routine returns, we reset the old bank and return control back to the Control Panel NDA (lines 46-77).

Every time the machine is booted up, the BootCDEV routine is called (lines 106-115). The loop is used to delay the boot process for about one second so the icon will stay on the screen a little longer than it normally would (this delay is not necessary and may be discarded if you so desire). When the delay is complete, we call a routine that verifies the existence of a TransWarp GS (line 110) and, if so, sets the speed to maximum (lines 111-115).

As soon as our icon is clicked on in the Control Panel NDA, a window is created with the size information we passed the NDA in the Rez source. The NDA then proceeds to call the CreateCDEV routine (lines 144-173) which again checks for the existence of a TransWarp GS. If the card is found, we use the window pointer passed to us by the NDA to create all the controls in the window (lines 161-173). If no card was found, a static text control is created that will notify the user of the card's absence (lines 151-158). The controls are not actually drawn until after the NDA calls InitCDEV.

InitCDEV retrieves the handles (lines 235-251) to the three controls that might be altered (for use by the SetControls routine), sets up the version number and maximum speed static text controls (lines 253-275), and calls SetControls to initialize the remaining controls. After all controls have been initialized and control is returned to the Control Panel NDA, it makes the controls visible and draws them.

At this time, the user can select an option from one of the pop-up menus. When one of the menus is "hit" (either by

clicking in it with the mouse or pressing the appropriate key equivalents) the Control Panel NDA calls our HitCDEV routine, passing to it the ID and the handle to the control. The first thing we do is verify that one of the menus was selected (lines 286-290) and, if so, call one of two routines (lines 291-294).

If the speed menu was selected, the SetSpeed routine determines which item was hit (lines 306-310), loads the speed index (lines 312-317), calls the TransWarp ROM routine SetCurISpeed (line 318), and calls SetControls to update the current speed static text control. Alternately, the SetIRQ routine follows the same process to determine if it should enable or disable the IRQ logic (lines 331-346). These routines aren't necessary for all CDEVs, but are required for the operation of the TransWarp CDEV.

SetControls determines if any of the operating parameters have changed since the last time through. First it retrieves the configuration word from the card and saves it (lines 392-394) and checks if the IRQ status has changed (lines 395-401). If the status has changed, it saves the new information, updates the IRQ menu, and redraws it (lines 402-409). The next step is to get the current speed index (line 412). Since future versions of the card might have more than three indices (0, 1, & 2) we ensure that the index is not greater than two (lines 413-415).

The reason for the code segment at lines 416-421 is that older versions of the TransWarp ROM returned a one here whether the card was running at fast or TransWarp speed. All that's done is if a one is returned as the index and the TransWarp flag is set in the configuration word, the index is made a two, else nothing is touched. Again, if the index has changed, the new value is saved, the speed menu is updated, and then redrawn (lines 428-435). The final task of SetControls is to determine whether the speed has changed or not. This segment works almost the same as setting the speed index except it uses the numeric speed returned by GetCurSpeed, instead of the speed index. If the returned value is 2600 and the TW flag is set, we make the speed 7000 (the TransWarp GS returns it's operating speed in kilohertz, so we convert the number to megahertz for displaying).

The RunCDEV routine (line 354) simply verifies that you have a TransWarp installed and, if so, calls SetControls (line 357) to determine if any operating parameters for the TransWarp have changed. This routine isn't necessary, but will automatically update the screen if you change the speed or IRQ setting from the TransWarp GS CDA.

The AboutCDEV routine creates a static text control in the window created by the NDA. The icon, CDEV name, author, version number, and OK button are automatically placed in the window by the Control Panel NDA, making life much easier on the programmer. I simply give a very brief description on what the CDEV does and display the key equivalents for the speed menu.

To put the entire CDEV together, follow these steps: enter the assembly source code (listing two) and type OA-6 to assemble and link it. If the assembly and link completed successfully, enter the Rez code (listing one) and the APW/ORCA command file (listing three) and execute it. The Rez code reads the control code and includes it in the CDEV. When the compile is done, the CDEV is automatically copied to the proper folder for testing. Select the Control Panel NDA from any desktop application, click on the TransWarp icon, and away you go.

Listing 1 - TransWarp GS CDev Rez Code

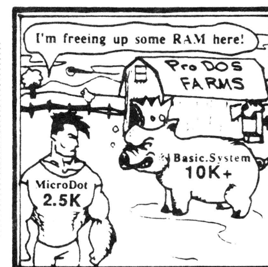
```
#include "types.rez"

resource rIcon (1) {
    0x8000,       /* color icon    */
    20,           /* Height        */
    28,           /* Width         */

/* image data */

    $"FFFFFFFFFFFFFFFFFFFFFFFFFFFF"
    $"F11111F1FFFFF1F11111F11111FF"
    $"FFF1FFF11FFF11F1FFFFF1FFFFFF"
    $"FFF1FFFF1F1F1FF1FF11F11111FF"
    $"FFF1FFFF11111FF1FFF1FFFFF1FF"
    $"FFF1FFFF11F11FF11111F11111FF"
    $"FFFFFFFFFFFFFFFFFFFFFFFFFFFF"
    $"FFFFFFFFF0000000000FFFFFFFFF"
    $"FFFFFFFFF0EEEEEEEE0FFFFFFFFF"
    $"FFFFFFFFF0EEEEEEEE0FFFFFFFFF"
    $"FFFFFFFFF0EEEEEEEE0FFFFFFFFF"
    $"FFFFFFFFF0000000000FFFFFFFFF"
    $"FFFFFFFF0DDDDDDDDDD0FFFFFFFF"
    $"FFFFFFF0DDDDDDDDDDDD0FFFFFFF"
    $"FFFFFFF00000000000000FFFFFFF"
    $"FFFFFFFFFFFFFFFFFFFFFFFFFFFF"
    $"FFFF444444444444FFFFFFFFFFFF"
    $"FFFFFFFFFFFFFFF444444444444F"
    $"F4444444444444FFFFFFFFFFFFFF"
    $"FFFFFFFFFFF4444444444444FFFF",

/* mask data */

    $"0000000000000000000000000000"
    $"0FFFFF0F00000F0FFFFF0FFFFF00"
    $"000F000FF000FF0F00000F000000"
    $"000F0000F0F0F00F00FF0FFFFF00"
    $"000F0000FFFFF00F000F00000F00"
    $"000F0000FF0FF00FFFFF0FFFFF00"
    $"0000000000000000000000000000"
    $"000000000FFFFFFFFFF000000000"
```

```
     $"000000000FFFFFFFFF000000000"
     $"000000000FFFFFFFFF000000000"
     $"000000000FFFFFFFFF000000000"
     $"000000000FFFFFFFFF000000000"
     $"00000000FFFFFFFFFFF00000000"
     $"0000000FFFFFFFFFFFFF0000000"
     $"0000000FFFFFFFFFFFFF0000000"
     $"000000000000000000000000000"
     $"0000FFFFFFFFFFFF00000000000"
     $"00000000000000FFFFFFFFFFFF0"
     $"0FFFFFFFFFFF000000000000000"
     $"00000000000FFFFFFFFFFFF0000"
};


read rCDEVCode (1,convert) "tw.code";
/* include the program */

resource rCDEVFlags (1) {
/* required flags
for all CDevs */
     wantAbout + wantCreate + wantHit + wantInit
+ wantBoot + wantRun,
     1,                          /* enabled  */
     15,                         /* version  */
     1,                          /* machine  */
     0,                          /* reserved */
     {0, 0, 85, 200},            /* rect.    */
     " TransWarp", /* name (15 chars. max)*/
     "Herb Hrowal & Palace Productions",
 /* Author (32 chars.max) */
     "v1.5"  /* Ver name (8 chars. max) */
};

/* "equates" for file */

#define Help              $1
#define NotFound          $2
#define SpeedMenu         $3
#define IRQMenu           $4
#define HelpText          $1001
#define NotFoundText      $2001
#define SpeedMenuItem1    $3001
#define SpeedMenuItem2    $3002
#define SpeedMenuItem3    $3003
#define IRQMenuItem1      $4001
#define IRQMenuItem2      $4002


resource rControlList (1) {
     {
     SpeedMenu,
     IRQMenu
     }
};


resource rControlTemplate (Help) {
     Help,
     {35,5,130,290},
     statTextControl {{
         NIL,
         fCtlProcNotPtr+RefIsResource,
         NIL,
         HelpText,
         Help
     }}
};

resource rTextForLETextBox2 (HelpText) {
  "The TransWarp CDEV allows you to change the"
  "settings of the TransWarp GS.  Speed Adj."
  "sets the system"
  " speed to normal, fast, or TransWarp. "
  "Appletalk"
  " IRQ enables or disables AT interrupts."
  TBEndOfLine
  TBEndOfLine
  "\$11N = Normal Speed"
  TBEndOfLine
  "\$11F = Fast Speed"
  TBEndOfLine
  "\$11T = TransWarp Speed"
};


resource rControlTemplate (SpeedMenu) {
 SpeedMenu,                 /* control ID */
 {23,10,35,190},            /* control rect */
 PopUpControl{{             /* control type */
    fType2PopUp,            /* flags */
    FctlProcNotPtr+FctlWantsEvents+RefIsResource,
/* MoreFlags */
    0,                      /* RefCon */
    0,                      /* Title Width */
    SpeedMenu,              /* Menu ref */
    SpeedMenuItem1          /* Initial Value */
    }}
};


resource rMenu (SpeedMenu) {
 SpeedMenu,              /* id ofmenu */
 RefIsResource * MenuTitleRefShift + RefIsRe-
source *ItemRefShift + fAllowCache,
 SpeedMenu,              /* id oftitle string */
 { SpeedMenuItem1,SpeedMenuItem2,SpeedMenuItem3
};  /* id's of items */
};

resource rPString (SpeedMenu) {
 "Speed Adj:   "
};

resource rMenuItem (SpeedMenuItem1) {
 SpeedMenuItem1,
 "N","n",          /* key equivalents */
 0,
 RefIsResource*ItemTitleRefShift+fXOR,
 SpeedMenuItem1
};

resource rPString (SpeedMenuItem1) {
 "Normal"
};


resource rMenuItem (SpeedMenuItem2) {
 SpeedMenuItem2,                           /* key
equivalents */
 "F","f",
 0,
```

```
 RefIsResource*ItemTitleRefShift+fXOR,
 SpeedMenuItem2
};

resource rPString (SpeedMenuItem2) {
 "Fast"
};


resource rMenuItem (SpeedMenuItem3) {
 SpeedMenuItem3,
 "T","t",                          /* key
equivalents */
 0,
 RefIsResource*ItemTitleRefShift+fXOR+fItalic,
 SpeedMenuItem3
};

resource rPString (SpeedMenuItem3) {
 "TransWarp"
};

resource rControlTemplate (IRQMenu) {
 IRQMenu,                    /* control ID */
 {40,10,52,190},             /* control rectangle */
 PopUpControl {{             /* control type */
    fType2PopUp,             /* flags */
    FctlProcNotPtr+RefIsResource,/* MoreFlags */
    0,                       /* RefCon */
    0,                       /* Title Width */
    IRQMenu,                 /* Menu ref */
    IRQMenuItem1             /* Initial Value */
    }}
};


resource rMenu (IRQMenu) {
 IRQMenu,  /* id of menu */
 RefIsResource * MenuTitleRefShift + RefIsRe-
source *ItemRefShift + fAllowCache,
 IRQMenu,  /* id of title string */
 { IRQMenuItem1, IRQMenuItem2 };/* id;s of items
*/
};

resource rPString (IRQMenu) {
 "AppleTalk IRQ:   "
};


resource rMenuItem (IRQMenuItem1) {
 IRQMenuItem1,
 "","",
 0,
 RefIsResource*ItemTitleRefShift+fXOR+fItalic,
 IRQMenuItem1
};

resource rPString (IRQMenuItem1) {
 "On"
};


resource rMenuItem (IRQMenuItem2) {
 IRQMenuItem2,
 "","",
 0,
 RefIsResource*ItemTitleRefShift+fXOR,
 IRQMenuItem2
};

resource rPString (IRQMenuItem2) {
 "Off"
};


resource rControlTemplate (NotFound) {
    NotFound,
    {20,20,35,180},
    statTextControl {{
        NIL,
        fCtlProcNotPtr+RefIsResource,
        NIL,
        NotFoundText,
        NotFound
    }}
};

resource rTextForLETextBox2 (NotFoundText) {
    "TWGS Not Installed.\n"
};
```


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

Listing 2 - TransWarp CDEV Source Code

```
  1       lst   off
  2       xc
  3       xc
  4       mx    %00
  5       typ   exe
  6       use   tw.macs  ;include macro file
  7       rel            ;make relocatable
  8       dsk   tw.code.l  ;assemble to disk
  9
 10 *========================================*
 11 *                                        *
 12 * Control code for the TransWarp CDev    *
 13 *                                        *
 14 *      Copyright (c) 1990                *
 15 *  Herb Hrowal, Palace Productions       *
 16 *      and Ariel Publishing              *
 17 *                                        *
 18 *========================================*
 19
 20 *------------------------------*
 21 *
 22 * Equates:
 23 *
 24
 25 CtlProcRefNotPtr = $1000
     ;more flag for extended controls
 26 TitleIsPtr =   0
     ;more flag for extended controls
 27
 28 TransWarpID =  $BCFF00
     ;location of ID. (TWGS)
 29 GetTWInfo =    $BCFF08
     ;returns with A = features, X = ver
 30 GetMaxSpeed =  $BCFF10    ;obvious
 31 Freq2Index =   $BCFF18
     ;converts the numerical speed to index
 32 GetCurSpeed =  $BCFF20    ;obvious
 33 GetCurISpeed = $BCFF28
     ;returns the current speed index
 34 SetCurISpeed = $BCFF2C
     ;sets the speed with the index
 35 DisableIRQLogic = $BCFF34
     ;kills IRQ interrupts
 36 EnableIRQLogic = $BCFF38
     ;enables IRQ interrupts
 37 GetTWConfig =  $BCFF3C
     ;returns info about the card
 38
 39
 40 *------------------------------*
 41 *
 42 * This is the routine called every time
 43 * an event occurs in my window
 44 *
 45
 46 Start
 47          phb
 48          phk
 49          plb          ;let's run in my bank
 50
 51          lda   5,s
     ;get all the pertinent data
 52          sta   Data2
 53          lda   7,s
 54          sta   Data2+2
 55          lda   9,s
 56          sta   Data1
 57          lda   11,s
 58          sta   Data1+2
 59          lda   13,s
 60          sta   Message
 61          lda   1,s ;move RTL address and bnk
 62          sta   11,s
 63          lda   3,s
 64          sta   13,s
 65          tsc
 66          clc
 67          adc   #10    ;fix the stack pointer
 68          tcs
 69
 70          lda   Message
 71          dec
 72          asl
 73          tax
 74          jsr   (Routines,x) ;do what NDA wants
 75
 76          plb               ;get original bank
 77          rtl               ;and return
 78
 79
 80 Routines da    Dummy       ;MachineCDev
 81          da    BootCDev
 82          da    Dummy       ;Reserved
 83          da    InitCDev
 84          da    Dummy       ;CloseCDev
 85          da    Dummy       ;EventsCDev
 86          da    CreateCDev
 87          da    AboutCDev
 88          da    Dummy       ;RectCDev
 89          da    HitCDev
 90          da    RunCDev
 91
 92 *------------------------------
 93
 94 Dummy    rts
 95
 96 *------------------------------*
 97 *
 98 * This called only at bootup. It displays
 99 * icon for CDev and anything else necessary
100 * for the CDev to run. Here making it delay
101 * the boot so icon will be displayed longer.
102 * It also sets the TWGS speed to max.
103 *
104
105 BootCDev
106          ldy   #20000      ;my delay
107 ]Loop    dey
108          bne   ]Loop
109
110          jsr   FindTransWarp
;go find the transwarp
111          lda   TWFlag
112          beq   :NoTW
113          jsl   SetCurISpeed
;set speed with index from Find
114
115 :NoTW    rts
116
117 *                          *
```

```
118 *
119 * This rtn called every time user clicks
120 * on Help btn. The title, author, & version
121 * # defined in REZ portion of the source.
122 *
123
124 AboutCDev
125          PushLong #0       ;space for result
126          PushLong Data1    ;window pointer
127          PushWord #2
     ;the control is a resource
128          PushLong #1       ;it is ID #1
129          _NewControl2      ;and call it
130          pla
     ;remove excess baggage
131          pla
132          rts
     ;and get outta here
133
134 *-----------------------------*
135 *
136 * This calls rtn to ck if TWGS is installed
137 * If it is, the controls are created in the
138 * window. If no TWGS exists then a msg for
139 * this condition is thrown in the window.
140 * (the MachinceCDev rtn should be used but
141 * is not yet supported by Cntrl Panel NDA).
142 *
143
144 CreateCDev
145          jsr   FindTransWarp
     ;go find the transwarp
146          lda   TWFlag
     ;let's us know if we found it
147          bne   :Ok
     ;yup, we did
148
149 * card not found so tell user and leave.
150
151          PushLong #0       ;space
152          PushLong Data1    ;wnd ptr
153          PushWord #2       ;reference a rsrc
154          PushLong #2       ;resource ID
155          _NewControl2      ;create it
156          pla
157          pla              ;remove the garbage
158          rts              ;and get outta here
159
160
161 :Ok      PushLong #0
162          PushLong Data1
163          PushWord #9
     ;reference a list of resources
164          PushLong #1       ;ID of list
165          _NewControl2
166
167          PushLong Data1
168          PushWord #3
     ;reference a list of pointers
169          PushLong #TextCtlList
     ;address of list
170          _NewControl2
171          pla
     ;pull the un-needed info from stack
172          pla
173          rts
174
175
176 TextCtlList    ;the static text control list
177          adrl  VersCtl
178          adrl  CurSpdCtl
179          adrl  MaxSpdCtl
180          adrl  0          ;end of list
181
182
183 VersCtl  dw    8             ;parm count
184          adrl  5             ;ctl ID
185          dw    8,25,20,190 ;bounds rect
186          adrl  $81000000   ;stat txt control
187          dw    0             ;flags
188          dw    CtlProcRefNotPtr+TitleIsPtr
        ;moreFlags
189          adrl  0             ;refcon
190          adrl  VersText    ;textref
191          dw    17            ;length of text
192
193 CurSpdCtl dw   8             ;parm count
194          adrl  6             ;ctl ID
195          dw    57,10,69,200 ;bounds rect
196          adrl  $81000000   ;stat txt control
197          dw    0             ;flags
198          dw    CtlProcRefNotPtr+TitleIsPtr
        ;moreFlags
199          adrl  0             ;refcon
200          adrl  CurSpdText ;textref
201          dw    28            ;length of text
202
203 MaxSpdCtl dw   8             ;parm count
204          adrl  7             ;ctl ID
205          dw    70,10,80,200 ;bounds rect
206          adrl  $81000000   ;stat txt control
207          dw    0             ;flags
208          dw    CtlProcRefNotPtr+TitleIsPtr
        ;moreFlags
209          adrl  0             ;refcon
210          adrl  MaxSpdText ;textref
211          dw    26            ;length of text
212
213 VersText asc   'TransWarp GS v'
214 VersNum  asc   'X.X',00
215
216 CurSpdText asc 'Current Speed = '
217 CurSpeed asc   'XX.XX Mhz.  ',00
218
219 MaxSpdText asc 'Maximum Speed = '
220 MaxSpeed asc   'XX.XX Mhz.',00
221
222 *-----------------------------*
223 *
224 * Init all controls that need it and
225 * get hndls to cntrls that might be changed
226 * throughout the program.
227 *
228
229 InitCDev
230          lda   TWFlag    ;is there a TWGS?
231          bne   :Ok       ;yup
232          rts             ;else no init req'd
233
234 :Ok
235          PushLong #0       ;space
236          PushLong Data1    ;window pointer
```

```
237          PushLong #3     ;ID for Speed Menu
238          _GetCtlHandleFromID
239          PullLong SMHandle ;store the hndl
240
241          PushLong #0       ;space
242          PushLong Data1    ;window pointer
243          PushLong #4       ;ID for IRQ Menu
244          _GetCtlHandleFromID
245          PullLong ATHandle ;store the hndl
246
247          PushLong #0       ;space
248          PushLong Data1    ;window pointer
249          PushLong #6   ;ID for Current Speed
250          _GetCtlHandleFromID
251          PullLong CSHandle ;store the hndl
252
253          jsl    GetTWInfo ;get version# in X
254          txa
255          ora    #$3030  ;make text printable
256          sep    $20       ;short A
257          sta    VersNum+2
     ;store the revision # in the string
258          xba              ;swap hi & lo acc
259          sta    VersNum
     ;store version # in the string
260          rep    $20       ;back to long A
261
262          PushWord MaximumSpd
     ;push max speed for the conversion
263          PushLong #MaxSpeed ;pointer to
string
264          PushWord #5       ;length of string
265          PushWord #0
     ;make it an unsigned number
266          _Int2Dec          ;convert it
267
268          sep    $20        ;short A
269          lda    MaxSpeed+3 ;make room for'.'
270          sta    MaxSpeed+4
271          lda    MaxSpeed+2
272          sta    MaxSpeed+3
273          lda    #'.'       ;decimal point
274          sta    MaxSpeed+2
275          rep    $20        ;long A
276
277          jmp    SetControls
     ;set remaining controls and return
278
279 *------------------------------*
280 *
281 * This is routine that is called every time
282 * the user selects a menu option.
283 *
284
285 HitCDev
286          lda    Data2      ;get controls ID
287          sec
288          sbc    #3
     ;we can only hit controls 3 & 4
289          cmp    #2         ;bad hit?
290          bge    :TooHigh
291          asl
292          tax
293          jsr    (:CtlRoutines,x)
     ;handle the control action
294 :TooHigh rts
295
296 :CtlRoutines
297          da     SetSpeed
     ;speed menu was hit
298          da     SetIRQ
     ;IRQ Menu was hit
299
300 *------------------------------*
301 *
302 * Find menu item hit & set speed accordingly
303 *
304
305 SetSpeed
306          PushWord #0        ;space
307          PushLong SMHandle
     ;handle for speed menu
308          _GetCtlValue
309          pla                ;get value of Ctl
310          sta    OldIndex
     ;store it so the menu doesn't get
311
     ; updated in the SetControls routine
312          and    #$F
     ;we only need the low nibble
313          dec
314          asl
315          tay                ;for indexing
316          lda    SpdIndex,y
317          tax
318          jsl    SetCurISpeed
     ;set the current speed
319
320          jmp    SetControls
     ;update changed controls and return
321
322 SpdIndex
323          dw     0,1,2
324
325 *------------------------------*
326 *
327 * Set the AppleTalk IRQ status of the card
328 *
329
330 SetIRQ
331          PushWord #0        ;space for result
332          PushLong ATHandle;hndl for IRQmenu
333          _GetCtlValue
334          pla
     ;get the controls value
335          sta    OldIRQ
     ;store so menu doesn't get
336
     ; updated in SetControls routine
337
338          and    #$F
     ;we only want low nibble
339          dec
340          beq    :IRQOn      ;turn on IRQ
341
342          jsl    DisableIRQLogic
343          rts
344
345 :IRQOn   jsl    EnableIRQLogic
346          rts
347
348 *------------------------------*
```

```
349 *
350 * This routine called every time my
351 * CDev gets a run event. It calls rtn to
352 * make sure cntrls display correct values
353
354 RunCDev
355          lda   TWFlag      ;TWGS in machine?
356          beq   :none       ;no
357          jsr   SetControls ;yes-update ctls
358 :none    rts
359
360 *-------------------------*
361 *
362 * Find out if TransWarp GS is installed.
363
364 FindTransWarp
365          stz   TWFlag      ;assume no card
366          ldal  TransWarpID ;get tID bytes
367          cmp   #'TW'       ;must match 'TWGS'
368          bne   :NoTWFnd
369          ldal  TransWarpID+2
370          cmp   #'GS'
371          bne   :NoTWFnd
372          sta   TWFlag ;found so make flag
373                      ; anything but zero
374          jsl   GetMaxSpeed ;get max speed
375          sta   MaximumSpd ;store for later
376          jsl   Freq2Index
    ;make index (returned in X)
377          cpx   #1
    ;old ROMs return a 1 here (wrong value)
378          bne   :Ok
379          ldx   #2
380 :Ok      stx   SpdIndex+4 ]
    ;store in index table
381 :NoTWFnd rts
382
383 *-------------------------*
384 *
385 * Ck cntrls that can be updated
386 * If changed, the control is redrawn
387 * with the correct value. If no change,
388 * this calldoes nothing.
389 *
390
391 SetControls
392          jsl   GetTWConfig
    ;get the configuration word
393          ldy   #1
394          sta   TWConfig   ;save it
395          and   #%0000_0000_0000_1000
    ;check the IRQ bit (0 = on)
396          beq   :On
397          ldy   #2
398 :On      tya               ;put it in A reg.
399          ora   #$4000
    ;make it a menu item ID
400          cmp   OldIRQ
    ;is it already set
401          beq   :SameIRQ
    ;it's already set. No need to change
402          sta   OldIRQ
    ;save for next pass
403
404          pha
405          PushLong ATHandle
    ;handle to control
406          _SetCtlValue
    ;and set the value
407
408          PushLong ATHandle
    ;push it again
409          _DrawOneCtl
    ;and draw it
410
411 :SameIRQ
412          jsl   GetCurISpeed ;get speed indx
413          cpx   #2
414          blt   :Cont
415          ldx   #2
    ;if it's more than 2, just make it 2
416 :Cont    cpx   #1
    ;if it's a 1 we might have to fix it
417          bne   :NoProb  ;for older cards.
418          lda   TWConfig
419          and   #%0000_0000_0000_0100
    ;is the TW flag set?
420          beq   :NoProb
421          ldx   #2
422
423 :NoProb  inx          ;bump it for item number
424          txa
425          ora   #$3000 ;make it menu item ID
426          cmp   OldIndex   ;already set?
427          beq   :SameIndex ;yup,no chg req'd
428          sta   OldIndex   ;save for nxt time
429
430          pha
431          PushLong SMHandle
    ;hndl for speed menu
432          _SetCtlValue     ;set the new value
433
434          PushLong SMHandle ;push it again
435          _DrawOneCtl      ;and redraw it
436
437 :SameIndex
438          jsl   GetCurSpeed
    ;get current speed in Khz
439          tax
440          cmp   #2600
    ;if it's 2600 we might need it fixed
441          bne   :NoFix
    ;nope, no problem then
442          lda   TWConfig
443          and   #%0000_0000_0000_0100
    ;is the TW flag set?
444          beq   :NoFix
    ;nope, false alarm
445          ldx   #7000
    ;it is an old card so make the
446 ;speed = 7 Mhz.
447 :NoFix
448          txa
449          cmp   OldSpeed   ;already set?
450          beq   :SameSpeed ;yup, just leave
451          sta   OldSpeed
    ;save it for nxt time through
452
453          pha
    ;push it for the conversion
454          PushLong #CurSpeed ;ptr to string
455          PushWord #5   ;length of string
```

```
456          PushWord #0
     ;we want an unsigned number
457          _Int2Dec           ;convert it
458
459          sep    $20         ;short A
460          lda    CurSpeed+3 ;make room dec pt
461          sta    CurSpeed+4
462          lda    CurSpeed+2
463          sta    CurSpeed+3
464          lda    #'.'        ;decimal point
465          sta    CurSpeed+2
466          rep    $20         ;long A
467
468          PushLong CSHandle
     ;hndl for text ctrl
469          _DrawOneCtl        ;draw it
470
471 :SameSpeed
472          rts
473
474 *------------------------------*
475 *
476 * Data area
477 *
478
479 Message   ds    2      ;msg passed to me
480 Data1     ds    4      ;data passed
481 Data2     ds    4      ;data passed
482 SMHandle ds    4      ;speed menu handle
483 ATHandle ds    4      ;IRQ menu handle
484 CSHandle ds    4
     ;current speed text control handle
485 TWFlag    ds    2      ;is there a TWGS?
486 OldSpeed ds    2
487 OldIndex ds    2
488 OldIRQ    ds    2
489 TWConfig ds    2      ;TW configuration word
490 MaximumSpd ds  2      ;maximum speed of card
```

Listing 3 - APW Make File

```
echo creating TransWarp CDev
compile tw.rez keep=TransWarp rez=(-t $c7)
duplicate TransWarp */system/cdevs/TransWarp
```

ZBasic Tricks

Squirreling Data Into Auxmem (& Elsewhere)

by Ross W. Lambert

I recently completed a very small contract programming job with some interesting implications for ZBasic programmers. Before I dig into it, though, I'd like to thank University of Wisconsin Professor Ron Myren for his willingness to let me share with y'all some of the things that I developed while under contract with him. Although I know that recent copyright law related court decisions have not made it necessary, I have nevertheless placed these routines under copyright in Ron's name. He has granted you folks (i.e. *8/16* subcribers) permission to use them, a gesture I find quite refreshing. Thanks, Ron.

Prof. Myren wrote a database in ZBasic that did not use the graphics modes - it was 80 column text only. Since ZBasic reserves the graphics pages, the good professor just *knew* there had to be away to access that extra 16K of memory .

The professor was quite correct. The only trouble is that ZBasic does not allow us to put arrays or other data into the graphics pages directly. That is good, in some cases, becase there is no hassle when you want to jump to a graphics mode. You can do so at any time and with no hesitation.

If you *never* want graphics, however, this situation is a waste of memory. And 16K is a lot of real estate on 128K Apples.

All of these things meant, therefore, that I'd have to manage the two 8K banks of memory on my own. As things turned out, this was not very difficult at all. In fact, the routines I developed provide the flexibility to manage the memory as either integer, floating point, or string arrays.

A little background: Professor Myren did not need blazing speed for retrieving a single element out of the extra memory space. Instead, he needed some zip for

moving an entire block of memory into the the extra array. The length of our array management routines was a consideration, too, hence the shorter the code the better. For those reasons, (and for flexibility's sake) I decided to use the built in ROM routine called AUXMOVE, called via some in-line MACHLG statements.

Call the AUXMOVErs

AUXMOVE just needs to know the address of the source data, the address of the destination, the number of bytes to move, and the direction of the transfer (i.e. auxilliary memory to main or main to aux). Because AUXMOVE will shuffle around the number of bytes you specify, it makes managing different types of arrays easier.

But let's begin at the beginning.

I decided to create a couple of "building block" functions before I got too far along. Since the project called for integer arrays, I called the two functions AuxPEEK_WORD and AuxPOKE_WORD. These little gems allow us to PEEK and POKE word sized values (two bytes) from and to aux mem at will. This is slightly dangerous, however, because so much of ZBasic proper lives over there. For our purposes, though, (PEEKing and POKEing around on the graphics pages) we are in pretty safe waters.

Admittedly, AuxPEEK_WORD and AuxPOKE_WORD do not do their jobs as fast as they could. However, because they use the protean AUXMOVE routine they can be changed to AuxPEEK_ELEMENT and AuxPOKE_ELEMENT quite easily. In such an instance, the length of your elements (and, hence, the amount of data moved about) would change to match the precision (i.e. number of bytes) of the floating point variable you were using or the length of the strings in your pseudo-array, etc.

The next function I created was FN BLOCKMOVE. I was very pleased at the speed with which AUXMOVE shuffles large blocks of memory around. This function would be useful if you wanted to move an entire array down into aux mem for storage, for example. By the way, if you want to find the address of any array, use VARPTR(Array(0)). And remember that ZBasic puts all of its variables in main memory. We are the ones getting tricky by shuffling them into aux mem.

After working out the first few functions, finishing things up was easy. FN XArray examines the parameters you pass and determines whether you want to read or write to main or aux mem. It will deposit any integer value you wish into any element of either the main bank array or the aux bank array.

Easy stuff.

Using FN XArray is even easier. Just think of the two banks of memory as you would any other array. If you want to have a look at the main mem array element number five, you can do so by calling the function like so:

```
IntValue = FN XArray (0,5,0,0)
```

In general, the syntax looks like this:

```
Int = FN XArray (Bank,Element,RdWrite,Val)
```

...where Bank is zero for main or a one for aux mem, Element is the element number in the array, RdWrite is a zero for read or a one for write, and Val is the integer to deposit into the array on a write operation. Val is ignored during a read, but the function itself returns the integer read.

One of the nice things about using a set of functions like these is that you don't need to calculate offsets into the range of memory you're using. The function does it for you. If you change the element size, however, the second line of the FN XArray function needs to be changed to:

```
Offset = Element * ElementSize
```

Note that the function will do its work mucho faster if you can use the bit-shift operators, changing the line above to something like:

```
Offset = Element << 4
```

This is not possible if you have odd sized array elements, but if speed is a very high priority, you may want consider wasting a byte per element. Bit level multiplication (i.e. in powers of 2) operates as much as 10 -20 times faster than standard integer multiplication. Of course, if speed is *that* important, you will want to write some custom assembly code that does not use AUXMOVE.

Variations on the theme

There's a jillion ways you can mold FN XArray to better suit your purposes. One thing I expect to do fairly soon is amend the function so that it manages both graphics pages as a *single* 16K array. This could be accomplished by having the function deposit even numbered elements in main mem, odd numbered elements in aux mem. Or, faster yet, the first 8K of elements goes in main mem, the second into aux mem.

Since I've had numerous questions of late regarding connecting assembly code and ZBasic, and since these functions may inspire you to try your hand at it, I thought I'd digress a tad and discuss some of the considerations in general terms. I'll get into more detail next month.

If you are interested in crafting custom assembly language code, keep in mind that your ZBasic program code is executing in auxilliary memory. This means that MACHLG statements - or, more accurately, the assembly code they represent - will be executing in aux mem, too. Furthermore, remember that your routine needs to be completely relocatable because there is no way to predict its final address.

One alternative to MACHLGs is the dimensioned buffering technique. Since ZBasic variables and arrays live in main memory, you can dimension an array equal to the size of your assembly routine, get its address with VARPTR, and then BLOAD the routine into that location. Don't forget to terminate your assembly stuff with an RTS and to avoid making any internal references. To use the code, just CALL Address.

If you don't want to BLOAD a routine, you can always convert the machine language hex codes to DATA statements and then POKE them into an array. Again, there are a few caveats: you can only POKE into main memory and the location of your array space is not easily nailed down. Your routine needs to be relocatable.

I am in the middle of experimenting with fixed position code deposited at the start of variable space ($AC00 less 1K for each file buffer). It seems to be working but I've not thoroughly tested it yet. Something tells me I'm playing with fire... my ZBasic tells me that the variable space starts at $A336, though the manual insists that it should be starting at $A400 ($AC00-$0800).

If you have a very short assembly routine, you can stick it at 768 in main memory. ZBasic does not use locations 768 - 975, hence it is also available for fixed position code.

The Future?

I love ZBasic. I develop ZBasic tools. This column evolved from my old *Znews* newsletter. But there comes a point when times and situations change. No, I'm not going to quit writing about ZBasic. But I wouldn't be doing my job if I blindly refused to consider other BASICs.

The reason is simple: Zedcor is a Macintosh company and is no longer actively developing Apple II software. ZBasic is good, but it will not get better, at least not right away. I can live with that - though I don't agree with it.

Micol Systems of Canada, however, has continued improving their BASICs, both GS and 8 bit. Frankly, I was not impressed with either version *initially*. But the Micol folks stuck with it (perseverance *counts*, my friends), and their BASICs (both 8 and 16 bit) have progressed a long way. There is now a lot to recommend them.

Two of the most significant advantages their 8 bit version has over ZBasic is a wonderful editor and local variables. Local variables make large scale and multi-programmer development significantly easier. To be honest and above board, I must tell you that I am now selling Micol's BASICs, so I am not particularly unbiased. In my own defense, I only sell products I believe in - I have to operate that way because of our unconditional money back guarantee on *all* of our products.

But this isn't an ad, so I'll click on the close box and close this window. After I do, I hope that you Z-fans find these functions - and the extra 16K of variable space they allow you to have - quite useful.

Bonus Code

In addition to my featured listing, the Extra Array Functions, I have tacked on a fun little routine called PROGRESS.FN. This is a little gizmo that will draw a thermometer-like scale in a box and update it as any long process happens.

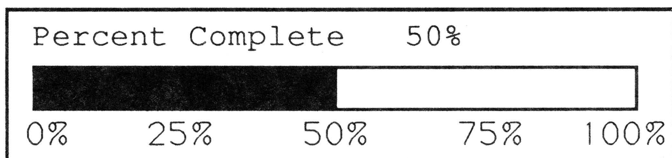
To use PROGRESS.FN, just insert it inside a loop or deposit it at selected points during long calculations, disk reads/writes, etc. I use it most when looping through and writing array elements to disk. It's fairly fast and doesn't slow down the process at all (well, at least not noticeably).

The syntax looks like this:

```
FN PROGRESS (X,Y,Amount,Total)
```

X and Y are the text screen coordinates of the upper left hand corner of the thermometer's box. Amount is the current status of the procedure being timed. That is, let's say you were writing 1000 strings to disk; Amount would be the current string you were working on. Total is the total number of strings, calculations, etc., that you needed to complete. In our 1000 string example above Total would be equal to 1000.

PROGRESS.FN displays a percentage, but you **don't** need to figure it out - the function calculates it for you automatically.



The PROGRESS.FN Thermometer Box

I hope you have some fun with it. PROGRESS.FN adds a very professional touch to your code without taking up very much space.

As will be my custom, I would like to close this column with the three "nevers" (with apologies to Chris Stasny):

One: Never get into an argument with a programmer about assemblers or pizza. You can't win, and you won't even get a byte.

Two: Never confuse a ringing phone with a CALL or a drug addict with a USR.

Three: Never , never, underestimate the power of BASIC.

== Ross ==

Extra Array FN listing

```
 REM ==============================
 REM ExtraArray FN
 REM
 REM by Ross W. Lambert
 REM a Work For Hire for
 REM Ron Myren, Univ of Wisconsin
 REM Copyright(C) 1989
 REM ==============================
  :
  :
 REM Equates for AuxMove (Apple nomencla-
ture, IIe Ref Man. p92)
  :
 A1L =&3C :REM source, low byte
 A1H =&3D :REM source, high byte
 A2L =&3E :REM end of source, low byte
 A2H =&3F :REM end of source, hi byte
 A4L =&42 :REM destination low byte
 A4H =&43 :REM destination high byte
  :
 LONG FN AuxPEEK_WORD (Address)
  REM source start
   POKE WORD A1L,Address
  REM source end (only 2 bytes)
   POKE WORD A2L,Address+1
  REM destination is IntVar
   POKE WORD A4L,VARPTR(IntVar)
  REM clc (means aux to main)
   MACHLG &18
  REM jsr AuxMove
   MACHLG &20,&11,&C3
 END FN = IntVar
  :
 LONG FN AuxPOKE_WORD (Address,Value)
  REM location of integer to POKE
   ValueAddr = VARPTR(Value)
  REM source start (lowbyte)
   POKE WORD A1L,ValueAddr
  REM source end (highbyte)
   POKE WORD A2L,ValueAddr+1
  REM destination
   POKE WORD A4L,Address
  REM sec (moves mem from main to aux)
   MACHLG &38
  REM jsr AuxMove
   MACHLG &20,&11,&C3
END FN
  :
  :
REM FN BlockMove - move chunks of memory
        between main & aux mem
  :
REM   * If Direction = 0, then the move will
        go MAIN to AUX
REM   * If Direction <> 0, then the move
        will go AUX to MAIN
  :
LONG FN BlockMove (Dir,Start,End,Dest)
   POKE WORD A1L,Start
   POKE WORD A2L,End
   POKE WORD A4L,Dest
   :
 REM aux to main
  LONG IF Direction = 1
    REM clc (move from aux to main)
    MACHLG &18
```

```
 XELSE:REM main to aux
   REM set carry...
   MACHLG &38
 END IF
  :
 REM jsr auxmove
 MACHLG &20,&11,&C3
END FN
:
LONG FN XArray (Bank,Element,Dir,Int)
  REM binary double (2 bytes each)
  Offset= Element << 1
  REM calculate actual spot in memory
  ElementAddr = 8192 + Offset
  :
  REM can¬t look or write outside 8K
  IF ElementAddr > 16383 THEN "Out"
  :
  REM write?
  LONG IF Direction = 1
   REM main mem?
   LONG IF Bank = 0
     REM if so, simple POKE WORD
     POKE WORD ElementAddr,Int
   XELSE    :REM nope, aux mem
     REM still simple!
    FN AuxPOKE_WORD (ElementAddr,Int)
   END IF
  XELSE:REM no, we READing from array
    LONG IF Bank = 0
     IntVar = PEEK WORD (ElementAddr)
    XELSE
   IntVar = FN AuxPEEK_WORD (ElementAddr)
    END IF
  END IF
"Out"
END FN = IntVar
:
:
REM Demo
:
Main = 0 : Aux = 1
Read = 0 : Write = 1
:
CLS
PRINT "To main:","To Aux:"
FOR Element = 0 TO 9
 FN XArray (Main,Element,Write,Element*2)
 FN XArray (Aux,Element,Write,Element*3)
 PRINT Element*2,Element*3
NEXT
:
PRINT "From Main:","From Aux:"
FOR Element = 0 TO 9
 VarMain = FN XArray(Main,Element,Read,0)
 VarAux = FN XArray (Aux,Element,Read,0)
 PRINT VarMain,VarAux
NEXT
:
:
PRINT "Press RETURN..."
INPUT R$ : CLS
PRINT "Speed test 1: Writing to main array
      1000 times..."
FOR Element = 0 TO 999
  FN XArray (Main,Element,Write,&FF)
NEXT
:
PRINT CHR$(7);"Done!"
PRINT
PRINT "Speed test 2: Writing to aux array
         1000 times..."
FOR Element = 0 TO 999
  FN XArray (Main,Element,Write,&FF)
NEXT
:
PRINT CHR$(7);"Done!"
:
PRINT
PRINT "Speed test 3: Reading from main
       array 1000 times..."
FOR Element = 0 TO 999
  FN XArray (Main,Element,Read,&FF)
NEXT
:
PRINT CHR$(7);"Done!"
PRINT
PRINT "Speed test 4: Reading from aux
array 1000 times..."
 FOR Element = 0 TO 999
  FN XArray (Aux,Element,Read,0)
NEXT
:
PRINT CHR$(7);"Done!"
PRINT
:
REM This is amazingly fast!!!
:
PRINT "Speed test 5: Moving a 2000 byte
        block from Aux mem to Main..."
FN BlockMove (1,8192,8192+2000,8192)
PRINT CHR$(7);"Done!"
PRINT
PRINT "This completes the test"
END
```

PROGRESS.FN Listing

```
REM _______________
REM Progress FN
REM
REM by Ross W. Lambert
REM Copyright (C) 1989
REM Most Rights Reserved
REM ======================
 :
 :
DIM 3 MouseText$,Norm$,Inverse$,4 PERCENT$
:
Inverse$ = CHR$(15)
MouseText$ = CHR$(27) + CHR$(15)
Norm$ = CHR$(14) + CHR$(24)
:
LONG FN PERCENT$(AMOUNT,TOTAL)
  PC! = AMOUNT\TOTAL
  PC! = PC! * 100
  PERCENT$ = STR$(PC!)
  PERCENT$ = LEFT$(PERCENT$,4) + "%"
END FN = PERCENT$
:
LONG FN Progress (AMOUNT,TOTAL,XPOS,YPOS)
  BoxLen = 53
  LONG IF FIRSTPASS = 0
    LOCATE XPOS+1,YPOS
    PRINT STRING$(BoxLen-1,"_")
    LOCATE XPOS+1,YPOS+4
    PRINT MouseText$;STRING$(BoxLen-1,"L")
    LOCATE XPOS,YPOS
```

```
    FOR ZY = YPOS+1 TO YPOS + 3
      LOCATE XPOS,ZY : PRINT "Z"
      LOCATE XPOS+BoxLen,ZY:PRINT "_"
    NEXT
    PRINT Norm$;
    LOCATE XPOS + 2,YPOS + 1:PRINT"Percent
       complete: ";FN PERCENT$(AMOUNT,TOTAL)
    LOCATE XPOS + 2,YPOS + 3
    PRINT " 0%        25%          50%
       75%        100%"
    FIRSTPASS = 1
  XELSE
    LOCATE XPOS + 3,YPOS +2
    BARLEN =(AMOUNT\TOTAL) * 50.0
    BAR$ = STRING$(BARLEN,CHR$(32))
   PRINT@(XPOS+2,YPOS+2)Inverse$;BAR$;Norm$
      LOCATE XPOS + 20,YPOS + 1:PRINT FN
          PERCENT$(AMOUNT,TOTAL)
       IF PERCENT$ = " 100%" THEN FIRSTPASS =
          0 : REM reset flag
     END IF
   END FN
   :
   :
   REM Demo
   :
   MODE 2
   :
   FOR X = 1 TO 50
     FN Progress (X,50,5,5)
   NEXT
   :
   LOCATE 1,20
   END
```

The Illusion of Motion

by Steven Lepisto

Editor: Animation is one of the most enjoyable and gratifying topics in programming. Remember the thrill you felt when you plotted your first high res shape on the 'ol II+? The IIgs has been a different story for most of us, however. Even simple shapes (let alone movement!) seem beyond reach.

For that reason I consider us doubly blessed to have two of the best GS animators in Appledom willing to teach us the art of animation. These gentlemen, Steven Lepisto and Chris McKinsey, will not only be sharing the "state of the art" with us, but since they have room to stretch out over several months - even years - we've also given them the freedom to take it step by step. Best of all, I bet they'd even be willing to handle some of your animation questions within their respective articles. == Ross ==

Animation. It is defined as the illusion of motion. It is the process of taking a series of unmoving images and making them appear to move. These articles will focus on how to do animation on the Apple IIgs computer from assembly language. It will start at the beginning with the basics, moving on to more and more complex concepts as we progress. I assume you have a working knowledge of 65816 machine language and a basic understanding of the IIgs's hardware. The basic concepts of animation are independent of the computer language used to implement them. However, since I happen to like working in assembly language, that is the language this article is geared towards. I use the Merlin-16+ assembler. I've attempted to code the program to minimize problems if you wish to convert the code to Orca/M or APW assemblers.

The program presented here is the start of several. We will build on this one program and it will give us a platform on which to experiment with different techniques of animation. In each new article, I will suggest some areas in which you can experiment on your own. I encourage your playing with the code as experience is the best teacher.

Goals of lesson 1:

1) Get a program up and running.

2) Get two images bouncing around the screen.

a) show how motion is achieved using velocities and positions.

b) show how to draw to the shires screen directly.

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

The first thing you need to do is type the program in. If you get the disk that goes with this issue, you are spared the typing. For the rest of you, type the program in. Before saving it, comment out the "use anim1.macs" line so we can build a macro file. Save the program under the name ANIMATION1. Now, from the editor's command box, type "mac ANIMATION1" and proceed to build the macros. The program uses the super macros supplied with Merlin-16+. When all the macros have been gathered, save the file under the name ANIM1.MACS. Then reload ANIMATION1 and restore the "use anim1.macs" line.

Assemble the program with open apple-6. If there are no errors, you can run it from the disk command menu with "=ANIMATION1". After a moment or two, you should see a blue diamond and green square zipping around a black screen. When you get tired of that, press a key or the mouse button to exit. We will use this procedure throughout the series to run the programs.

What's It Doing?

Motion of the images is achieved by the use of a change in position applied as a vector (a vector is a line with a direction). This change in position vector is called velocity. To move the image, you add the veolicty to the current position of the image to get a new position. You then erase the image from the old position and redraw it at the new position. In the program, the position of the image is represented by an X,Y coordinate (Y indicates which row and X indicates which pixel on that row the image is on). The velocity is broken up into two components: the horizontal or X velocity and the vertical or Y velocity. By adding the components of the velocity to the proper part of the coordinates, the image can move in any direction.

The coordinates used in the program have the origin (where X and Y are both 0) at the upper left corner of the screen. Incrementing X moves to the right and incrementing Y moves down. So for example, to move the image left one pixel, you add to the X coordinate an X velocity of -1 and to the Y coordinate you add a Y velocity of 0 (it isn't moving in that direction). To move the image at a 45 degree angle (down and to the right one pixel), you add an X velocity of 1 to the X coordinate and a Y velocity of 1 to the Y coordinate.

When the image hits a boundary, it is bounced off in a predictable way. This bouncing is achieved by inversing

the appropriate velocity (converting it from negative to positive and vice versa). If the image hits the left or right boundaries, inverse the X velocity. If the image hits the top or bottom boundaries, inverse the Y velocity. This causes the image to reflect from the boundary hit as you would expect it to. To see the code for this, look at the routine MOVE_IMAGES. I first add the X velocity to the X coordinate then check that new position against the left and right boundaries. If the image hits a boundary, I inverse the X velocity and set the new position to that boundary. This way, the image looks like it hits the boundary.

One thing to note here is by adding the velocity to the position to get a new position, the image doesn't move through all intervening positions: it jumps to the new position. This is why I set the image to the boundary that was hit; otherwise if the velocity was large enough it might look like the image bounced off an invisible barrier in front of the boundary instead of the boundary itself.

If you make the jumps small enough the eye sees smooth motion. To give the illusion of greater speed, you can increase the size of the jumps. However, if the jump is too large, the eye will not be able to track the image well and it will look like it is skipping or stuttering. In the program, the range of the horizontal and vertical velocities is 0, 1, and 2. This represents the number of pixel positions to jump each time MOVE_IMAGES is called. The reason for the limit will be explained a little later. If you do increase the velocities beyond 2, the images will leave a trail behind them.

So, by adding an X velocity to the X coordinate, you cause the image to move left or right and by adding a Y velocity to the Y coordinate, you cause the image to move up or down. Adding both velocities at the same time causes the image to move in other directions. By increasing the velocity, you increase the apparent speed of the image. That's pretty much all there is to moving an image on the screen.

How's It Doing It?

Animation on a computer is generally not a simple thing. The basic steps of motion are:

1) Draw image at the specified coordinates
2) Update the coordinates with the velocity
3) Erase the image from the original coordinates
4) Draw the image at the new coordinates

This process needs to be done in such a way that the eye sees only the change in position of the image (which gives the illusion of motion). If the erasing step becomes visible, the eye will see that and the illusion will be interrupted. This phenomenon is called flicker. The goal of animation on the computer is to eliminate flicker for flicker is a Bad Thing (unless you want it for an effect—for now, we don't).

There are many techniques for eliminating flicker and we will discuss several of them in coming installments. The technique I'm using in the program is the simplest: combine the erasing and drawing steps so the image is never completely invisible to the eye. This technique only works on a background that is mono-colored (in this case, black). Notice how the images are defined in the source code. See the border of 0's around the images? That border is the same color as the background so if you draw the image then redraw it shifted to one side, the border will erase that part of the old image not overlapped by the new. This is how to combine the erase and the draw.

There are advantages and disadvantages to this method.

The advantages are:

1) Speed. Things go a lot faster if you don't have to worry about preserving the background.

2) The amount of memory needed to store the graphics can be lessened significantly over other techniques. For example, one technique for preserving the background requires the use of an image mask which eliminates the erasing border by making it essentially transparent (and yes, this does mean another way is needed to erase the background). However, having a mask for every image will double the amount of memory required to hold the images.

The disadvantages are:

1) Images can't overlap. If they do, the border not only erases the old image but the image being overlapped. You can see this occasionally as the blue diamond passes over the green square.

2) The background must be the same color as the erasing border (or vice versa). This tends to limit the

background graphics quite a bit.

3) You can't use too large of a velocity otherwise the image will be displaced too far for the erasing border to have an effect. And if you want large velocities, you can add a larger erasing border but then you start spending more and more time drawing the image to the screen as it gets larger and larger and more and more of that time is spent in erasing. You need to draw the line at some point. In this program the width and height of the erasing border is 2 pixels. This is why you shouldn't increase the velocities in the program past 2: you will displace the image more than two pixels and the erasing border won't work.

In Conclusion

So, that's all this program is really doing: adding velocity components to coordinates and drawing images at those coordinates. This is the essence of animation, the Illusion of Motion.

Things To Experiment With

There are a number of things to play with in this program. By all means, feel free to experiment beyond the suggestions.

1) In the routine ANIMATE, there is a call to PAUSE_A_MOMENT. Increase or decrease the value being loaded into the A register to slow down or speed up the overall motion. Set the A register to 0 to get the fastest motion. Slowing down the action can get you a better look at the images' motions.

2) If you wish, you can try adding more of the basic images to the display. First, modify the constant MAXIMAGES to the number of images you want. Then, at the end of INIT_IMAGES, modify the default arrays to add more images. Make sure you don't add more images than are allowed by MAXIMAGES. Also, make sure every default value is filled in for all images.

3) Try changing the motion boundaries to smaller values. Be sure the default starting positions of the images are within the motion boundaries (the images won't erase properly the first time through if they start outside the boundaries).

4) If you are feeling really ambitious, try adding a whole new image by "drawing" it in source code. Don't forget to modify the default arrays to reflect the size of the new image. And make sure the image is an even number of bytes wide.

5) Finally, notice how I determine the location and size of the super hires screen. This technique allows me to easily adjust to changing conditions. It even allows me to run the program in 640 mode if the images were modified properly.

Listing 1 - Illusions of Motion

```
 1          lst   off
 2          rel
 3          typ   s16
 4          dsk   animation1.1
 5
 6          xc
 7          xc                  ;65816 mode
 8          mx    %00           ;full 16-bit regs
 9          cas   in
10          use   anim1.macs
11
12 *=========================================
13 * Illusions of Motion
14 * by Stephen P. Lepisto
15 * date: 1/22/90
16 * Assembler: Merlin-16+ v4.08+
17 *=========================================
18
19 * Direct page definitions
20
21          dum   $00
22 deref_ptr ds   4
23 screen_ptr ds  4
24 rowadrs_table ds 4
25 image_ptr ds   4
26          dend
27
28 * # of images that can be handled
29
30 MAXIMAGES =    2
31
32 *=========================================
33
34 Start    phk
35          plb
36          jsr   dostartup
37          bcs   shutdown      ;err in startup
38          jsr   Animate
39 shutdown jsr   doshutdown
40          _Quit quitparms
41          brk   $de           ;shouldn't break
42
43 *----------------------------
44 * Main loop where it all happens.
45
46 Animate  jsr   init_images;init image arrays
47          jsr   init_boundaries ;for motion
48 :event_loop jsr draw_images ;draw all images
49          jsr   move_images ;move all images
50          lda   #1            ;do a short pause
```

```
 51          jsr   pause_a_moment
 52          jsr   read_key
 53          bcc   :event_loop ;no key pressed
 54          rts
 55
 56 *-----------------------------------
 57 * Apply velocities to images to make move.
 58 * Bounce off motion boundaries as needed.
 59
 60 move_images stz image_index
 61 :move_loop lda image_index
 62          asl
 63          tax
 64          lda   xposition,x
 65          clc
 66          adc   xvelocity,x
 67          bmi   :1          ;way past left
 68          cmp   left_boundary
 69          bcs   :2          ;not on left edge
 70 :1       jsr   invert_xvel ;else bounce it
 71          lda   left_boundary
 72          bra   :2
 73 :2       pha
 74          clc
 75          adc   image_width,x
 76          cmp   right_boundary
 77          pla
 78          bcc   :3          ;not on right edge
 79          jsr   invert_xvel ;else bounce it
 80          lda   right_boundary
 81          sec
 82          sbc   image_width,x
 83 :3       sta   xposition,x
 84
 85          lda   yposition,x
 86          clc
 87          adc   yvelocity,x
 88          bmi   :4          ;way above top
 89          cmp   top_boundary
 90          bcs   :5          ;below top edge
 91 :4       jsr   invert_yvel ;else bounce it
 92          lda   top_boundary
 93          bra   :6
 94 :5       pha
 95          clc
 96          adc   image_height,x
 97          cmp   bottom_boundary
 98          pla
 99          bcc   :6          ;above bottom edge
100          jsr   invert_yvel ;else bounce it
101          lda   bottom_boundary
102          sec
103          sbc   image_height,x
104 :6       sta   yposition,x
105          inc   image_index
106          lda   image_index
107          cmp   number_of_images
108          bcc   :move_loop
109          rts
110
111
112 * Invert X velocity for illusion of bounce.
113
114 invert_xvel lda xvelocity,x
115          eor   #$ffff
116          inc
117          sta   xvelocity,x
118          rts
119
120 * Invert Y velocity for illusion of bounce.
121
122 invert_yvel lda yvelocity,x
123          eor   #$ffff
124          inc
125          sta   yvelocity,x
126          rts
127
128 *-----------------------------------
129 * Draw all images at current pos on screen.
130
131 draw_images stz image_index
132 :draw_loop lda image_index
133          asl
134          tax
135          asl
136          tay
137          lda   image_bytewidth,x
138          sta   plot_bytewidth
139          lda   image_height,x
140          sta   plot_height
141          lda   xposition,x
142          sta   plot_xpos
143          lda   yposition,x
144          sta   plot_ypos
145          lda   image_adrs,y
146          sta   image_ptr
147          lda   image_adrs+2,y
148          sta   image_ptr+2
149          jsr   plot_image
150          inc   image_index
151          lda   image_index
152          cmp   number_of_images
153          bcc   :draw_loop
154          rts
155
156 *-----------------------------------
157 * Set up motion boundaries to shires screen.
158
159 init_boundaries
160          stz   left_boundary
161          lda   shires_width
162          sta   right_boundary
163          stz   top_boundary
164          lda   shires_height
165          sta   bottom_boundary
166          rts
167
168 *-----------------------------------
169 * Initialize variables showing & moving
170 * images across the screen.
171
172 init_images ldx #0
173 :1       lda   def_velx,x
174          sta   xvelocity,x ;X velocity
175          lda   def_vely,x
176          sta   yvelocity,x ;Y velocity
177          lda   def_posx,x
178          sta   xposition,x ;starting X
179          lda   def_posy,x
180          sta   yposition,x ;starting Y
181          lda   def_width,x
182          sta   image_width,x ;in pixels
```

```
183          lda     def_bytewidth,x
184          sta     image_bytewidth,x ;in bytes
185          lda     def_height,x
186          sta     image_height,x ;in scan lines
187          txa
188          asl
189          tay
190          lda     def_image,y
191          sta     image_adrs,y ;addr of image
192          lda     def_image+2,y
193          sta     image_adrs+2,y
194          inx
195          inx
196          cpx     #MAXIMAGES*2
197          bcc     :1
198          txa
199          lsr
200          sta     number_of_images
201          rts
202
203 *Defaults. Note that velocities should never
204 *be > 2: images will leave trails otherwise.
205
206 def_velx da     1,-2
207 def_vely da     1,-2
208 def_posx da     4,300
209 def_posy da     10,100
210 def_width da    16,16
211 def_bytewidth da 8,8
212 def_height da   15,15
213 def_image adrl basic_image_1,basic_image_2
214
215 *------------------------------
216 * Put byte-oriented image onto shires scrn.
217 * Assumes image is even # of bytes wide.
218
219 plot_image lda plot_ypos
220          asl                ;Y -> index
221          tay
222          lda     plot_xpos
223          lsr                ;pixels to bytes
224          clc
225          adc     [rowadrs_table],y
226          sta     screen_ptr
227          lda     shires_adrs+2
228          sta     screen_ptr+2
229 :row_loop ldy    #0
230 :byte_loop lda   [image_ptr],y
231          sta     [screen_ptr],y
232          iny
233          iny
234          cpy     plot_bytewidth
235          bcc     :byte_loop
236          lda     image_ptr
237          clc
238          adc     plot_bytewidth
239          sta     image_ptr
240          bcc     :1
241          inc     image_ptr+2
242 :1       lda     screen_ptr
243          clc
244          adc     shires_byte_width
245          sta     screen_ptr
246
247          dec     plot_height
248          bne     :row_loop
```

```
249          rts
250
251 *------------------------------
252 * Find size of shires screen and where it is.
253 * Also, determine where row address table is.
254 * This rtn takes advantage of default port
255 * when QuickDraw first starts up.
256
257 plot_setup ~GetPortLoc #shireslocinfo
258          ~GetAddress #1
259          pulllong rowadrs_table
260          rts
261
262 *------------------------------
263 * Start up some tools needed for program.
264 *
265 * Output:
266 *   carry: set if some sort of error occured
267
268 dostartup _TLStartUp
269          _MTStartUp
270          ~MMStartUp
271          pla
272          sta   ProgramID
273          clc
274          adc   #$100
275          sta   PrivateID
276
277 * Get direct page for QuickDraw ($300 worth).
278
279          ~NewHandle #$300;ProgramID;#$c015;#0
280          pulllong deref_ptr
281          bcs   :x
282          lda   [deref_ptr]
283          pha
284          pushword #0       ;320 mode
285          pushword #0       ;screen width
286          pushword ProgramID
287          _QDStartUp
288          bcs   :x
289          jsr   plot_setup
290          clc
291 :x       rts
292
293 * Shut down tools started and dispose of any
294 * private memory we may have allocated.
295
296 doshutdown _QDShutDown
297          ~DisposeAll PrivateID
298          ~MMShutDown ProgramID
299          _MTShutDown
300          _TLShutDown
301          rts
302
303 *------------------------------
304 * this fn returns ascii key val if available
305 * else it returns a zero.
306 *
307 * Output:
308 *   Carry: set if key pressed (key in A.reg).
309
310 keyboard =      $e0c000
311 keystrobe =     $e0c010
312
313 read_key sep    #$20
314          ldal   keyboard
```

```
315          bpl   :x
316          stal  keystrobe
317 :x       rep   #$20
318          and   #$ff
319          cmp   #$80
320          rts
321 ----------------------------
322 *
323 * Do a short pause.
324 *
325 * Input:
326 *  A.reg:# of 5000th sec intervals to wait.
327
328 gs_speed_control = $e0c036
329
330 pause_a_moment
331          tax
332          beq   :x          ;0=no delay
333          sep   #$20
334          ldal  gs_speed_control
335          and   #$80
336          sta   old_speed
337          ldal  gs_speed_control
338          and   #$7f
339          stal  gs_speed_control
340          rep   #$20
341 :wait    jsr   wait
342          dex
343          bne   :wait
344          sep   #$20
345          lda   old_speed
346          oral  gs_speed_control
347          stal  gs_speed_control
348          rep   #$20
349 :x       rts
350
351 * Typical wait loop, waits for .005 sec
352
353 wait     sep   #$20
354          lda   #42
355 :1       pha
356 :2       sec
357          sbc   #1
358          bne   :2
359          pla
360          sec
361          sbc   #1
362          bne   :1
363          rep   #$20
364          rts
365 ----------------------------
366 *
367 * variables.
368
369 quitparms adrl 0
370          da    $0000       ;not restartable
371 ProgramID ds   2
372 PrivateID ds   2
373 number_of_images ds 2      ;# of images
374 image_index ds 2           ;loop index
375 old_speed ds   2
376
377 * Motion boundaries (in pixels)
378
379 left_boundary ds 2
380 right_boundary ds 2
381 top_boundary ds 2
382 bottom_boundary ds 2
383
384 * Plot_image inputs.
385
386 plot_ypos ds   2
387 plot_xpos ds   2
388 plot_height ds 2
389 plot_bytewidth ds 2
390
391 * Screen location record.
392
393 shireslocinfo
394 portSCB  ds    2
395 shires_adrs ds 4
396 shires_byte_width ds 2
397          ds    4           ;space filler
398 shires_height ds 2
399 shires_width ds 2
400
401 * Image arrays
402
403 xposition ds   MAXIMAGES*2
404 yposition ds   MAXIMAGES*2
405 xvelocity ds   MAXIMAGES*2
406 yvelocity ds   MAXIMAGES*2
407 image_height ds MAXIMAGES*2
408 image_width ds MAXIMAGES*2
409 image_bytewidth ds MAXIMAGES*2
410 image_adrs ds  MAXIMAGES*4
411
412 * ----------------------------
413 *Basic images, ea 16 pxls(8 bytes)wide by 15
414 *lines hi.Plot_Image assumes even byte width
415
416 basic_image_1 hex 0000000000000000
417          hex   0000000000000000
418          hex   000aaaaaaaaaa000
419          hex   00aaaaaaaaaaaa00
420          hex   00aaaaa00aaaaa00
421          hex   00aaaa0000aaaa00
422          hex   00aaa000000aaa00
423          hex   00aa00000000aa00
424          hex   00aaa000000aaa00
425          hex   00aaaa0000aaaa00
426          hex   00aaaaa00aaaaa00
427          hex   00aaaaaaaaaaaa00
428          hex   000aaaaaaaaaa000
429          hex   0000000000000000
430          hex   0000000000000000
431
432 basic_image_2 hex 0000000000000000
433          hex   0000000000000000
434          hex   0000000440000000
435          hex   0000004444000000
436          hex   0000044444400000
437          hex   0000444444440000
438          hex   0004444444444000
439          hex   0044444444444400
440          hex   0004444444444000
441          hex   0000444444440000
442          hex   0000044444400000
443          hex   0000004444000000
444          hex   0000000440000000
445          hex   0000000000000000
446          hex   0000000000000000
```

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 **T**hingamajigs) 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* •

Tom Weishaar said it best in the October '88 edition of *Open-Apple* (now *A2-Central*):

"[This] ...is one of thoese rare educational software packages that does things in the classroom with a computer that can't be done any other way. It's the foundation for a semester long course... Like all the best educational software, Shem the Penman's Guide comes with a student manual on disk, where it can be shortened, lengthened, or otherwise modified..."

Tom forgot to mention that this is a terrific introduction to writing interactive fiction for programmers, too. 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*™ •

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 $39.95 (5.25" and 3.5" disks supplied).

NOTE: If you are already a ProTools owner, be sure and send us a blank disk and a SASE so that we can give you your free update. The new additions and bug fixes make it very worthwhile!

• *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* •

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..........$69.95 MAB GS.............$99.95

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.

Them's the BRKs:
Relocation in 8 bit assembly

by Jerry Kindall, 8-bit Editor

I had the pleasure of meeting Bob Sander-Cederlof at the A2-Central Developer Conference last July in Kansas City. Bob brought with him to the conference a number of leftover *Apple Assembly Lines* back issues, and was giving them away to anyone who wanted them. In the resulting feeding frenzy, I managed to snatch a nearly complete set. I'd been a deprived child; I'd heard of AAL but never actually seen an issue. Now I know what I was missing.

One interesting article, published way back in June of 1982, was entitled "Implementing New Opcodes Using BRK" and demonstrated how to use the BRK opcode to tie machine-language routines into your programs as if they were new opcodes. The three new opcodes described in the article were designed to help you to write completely relocatable code. However, the BRK handler was required to be at a fixed address (instead of being relocatable along with the rest of the program), and the three new instructions, while useful, offered only relative JMP, JSR, and LEA (load effective address).

I thought it would be fun to generalize Bob's idea into a multipurpose relocation routine. (OK, so my idea of "fun" is a little strange.) My variation on this theme allows you to turn any three-byte 6502 opcode into a relocatable instruction by preceding it with a BRK opcode. The run-time relocator can also handle immediate-mode instructions that load a register with an address, an addressing mode which has traditionally been the bane of relocatable programming.

Add some macros, and it's almost as if the 6502 suddenly sprouted a whole new relative addressing mode. Programs written with the run-time relocator can be loaded and run at ANY address, without modification.

The Code

Rather than trying to start out by explaining how the program works and how to use it, I'll give you the code first and then try to explain it. There's no need to assemble this code; it's designed to be used as a Merlin PUT file. Save it to disk as RTR.

Listing 1

```
1              lst  on
2   ***********************************************
3   *****                                     *****
4   *****         Run-Time Relocator          *****
5   *****          by Jerry Kindall           *****
6   *****                                     *****
7   *****             for 8/16                *****
8   *****         Merlin 8 Assembler          *****
9   *****                                     *****
10  ***********************************************
11             lst  off
12
13             org  $800
14
15  *Install BRK handler by finding current run
16  *addr of this routine and storing it into
17  *the BRK vector at $3F0-$3F1
18
19  RTR        php              ;save ress on stack
20             pha
21             txa
22             pha
23             tya
24             pha
25             lda  $3F0
        ;save current BRK vector
26             pha
27             lda  $3F1
28             pha
29  :getadr    jsr  $FF58           ;call known RTS
30             tsx
        ;un-pop address onto stack
31             dex
32             dex
33             txs
34             pla
        ;figure low byte of BRK handler
35             clc
36             adc  #RTR_PROC-:getadr-2
37             sta  $3F0
38             pla
        ;figure hi byte of BRK handler
39             adc  #/RTR_PROC-:getadr-2
40             sta  $3F1
41             pla
        ;get old BRK vector
42             brk
        ; and save in hold area
```

```
43          sta  RTR_OLDB+1
44          pla
45          brk
46          sta  RTR_OLDB
47          pla
      ;restore all registers
48          tay
49          pla
50          tax
51          pla
52          plp
53          brk
54          jmp  RTR_CONT
      ;continue with program
55
56  RTR_OLDB  ds  2
      ;hold area for old BRK vector
57
58 * Process BRK reqst by relocating 3-byte
59 * instruction following the BRK.
60 *
61 *Special case opcodes: BRK (00)= a real BRK,
62 *NOP (EA) = immediate mode relocation
63
64  RTR_PROC  sec
      ;point $3A to BRK
65          lda  $3A
66          sbc  #2
67          sta  $3A
68          bcs  :0
69          dec  $3B
70  :0      ldy  #1
71          lda  ($3A),y
      ;get opcode
72          beq  :brk
      ;it's a REAL BRK
73          cmp  #$EA
74          bne  :adjust
75
76  * Adjust 2 immediate (2-byte) instructions
77
78          iny
79          iny
80          sec
      ;subtract out assembly address
81          lda  ($3A),y
82          sbc  #:proc
83          tax
      ;save low byte
84          iny
      ;subtract out hi byte
85          iny
86          lda  ($3A),y
87          sbc  #/:proc
88          pha
      ;save hi byte
89          dey
      ;add in runtime address
90          dey
91          clc
92          txa
93          adc  $3F0
94          sta  ($3A),y
      ;modify low byte
95          iny
96          iny
97          pla
      ;add in hi byte
98          adc  $3F1
99          sta  ($3A),y
      ;modify hi byte
100         clc
101         bcc  :goback
102
103 * Handle BRK instruction
104
105 :brk      jmp  $FA59        ;do a REAL BRK
106
107 * Adjust absolute (3-byte) instruction
108
109 :adjust   iny
110         sec
      ;subtract out assembly address
111         lda  ($3A),y
112         sbc  #:proc
113         tax
      ;save low byte
114         iny
      ;subtract out hi byte
115         lda  ($3A),y
116         sbc  #/:proc
117         pha
      ;save hi byte
118         dey
      ;add in runtime address
119         clc
120         txa
121         adc  $3F0
122         sta  ($3A),y
      ;modify low byte
123         iny
124         pla
      ;add in hi byte
125         adc  $3F1
126         sta  ($3A),y
      ;modify hi byte
127
128 :goback   ldy  #0
      ;NOP out the BRK
129         lda  #$EA
130         sta  ($3A),y
131         jsr  $FF3F
      ;restore registers
132         jmp  ($3A)              ;go do
the instruction
133
134 * Routine to deinstall Run-Time Relocator
135 * Call using BRK followed by JSR RTR_DINS
136 * No registers are changed
137
138 RTR_DINS  php
139         pha
140         brk
141         lda  RTR_OLDB
142         sta  $3F0
143         brk
144         lda  RTR_OLDB+1
145         sta  $3F1
146         pla
147         plp
```

```
148          rts
149
150  RTR_CONT
151          lst  on
```

Using The Relocator

To include the run-time relocator in your own programs, all you need to do is include a PUT RTR as the first statement in your source code. Do NOT include a REL or ORG statement. If you wish to assemble to disk, include the DSK psuedo-op before the PUT RTR.

In your program, insert a BRK opcode before each absolute or absolute indexed instruction which references an address within your code. You can insert a BRK before ANY three-byte instruction — JSR, JMP, EOR, LDA, or even BIT. Here's a simple example using this capability.

Listing 2

```
 1          put  rtr
 2
 3  cout     =    $FDED
 4
 5  start    ldx  #0        ;set index to zero
 6           brk
       ;relocate next instruction
 7  loop     lda  text,x    ;get character
 8           beq  exit      ;end of string
 9           jsr  cout      ;print character
10           inx         ;pt to next character
11           bne  loop
       ;always (when string <255 chars)
12  exit     rts
13
14  text     asc  "This is an example run-time
       relocation program."
15           hex  8D00
```

Check out the BRK instruction in line 6. When that line is executed, the run-time relocator gets control and patches the next statement to work at the program's current address. The BRK opcode is replaced with a NOP so the relocator won't erroneously adjust the already-corrected instruction the next time it's executed.

Line 9's JSR is NOT preceded by a BRK instruction. That's because COUT is not a location within your program; it's an absolute ROM location. You definitely don't want to mess with the address of a call to the ROM. Only instructions which reference locations within your program should be preceded by a BRK. Also remember that branching instructions (BNE, etc.) are already relocatable and do not need to be prefaced with a BRK.

Note that the BNE instruction in line 11 branches back to line 7, not line 6. The first time line 6 is executed, it'll be replaced with a NOP instruction — so there's no reason to branch back to a NOP the next time through.

Immediate Mode Instructions

As I mentioned earlier, the run-time relocator has the ability to adjust immediate-mode instructions as well. Here's a short example of that:

Listing 3

```
 1          put  rtr
 2
 3  prntax   =    $F941
 4  cout     =    $FDED
 5
 6  start    lda  #"$"      ;print dollar sign
 7           jsr  cout
 8           brk            ;relocate
 9           nop            ;effective address
10           lda  #rtr
       ;get address of relocator
11           ldx  #/rtr
12           jsr  prntax    ;print in hex
13           lda  #$8D      ;print CR
14           jsr  cout
16           rts
```

The purpose of the above code is to print the run-time address of the program (in hex notation) on the screen. Note that line 8's BRK is followed in line 9 by a NOP. This is a special flag to the run-time relocator that two immediate mode instructions follow. This is similar to the "effective address" mode on some other processors. (In fact, I chose NOP as the effective address flag because its hex code is EA — an acronym for effective address. Some might call this pretzel logic.)

In line 10, we load the Accumulator with the low byte of the label RTR; in line 11, the high byte of the same label is loaded into the X register. The label RTR is defined in the relocator source code (see line 18 of the first source listing) as the beginning of the executable code. The run-time relocator will assume that the operand of the first instruction is the low byte of an address, and the operand of the second is the high byte of the same address. So this sequence of instructions loads the Accumulator and the X-register with the address of the first instruction in the program, adjusted for the program's run-time address.

Notice that (in line 13) we load the accumulator with a constant number. $8D is the code for a carriage return, and should remain constant no matter where in memory the program runs.

This feature of the run-time relocator is designed for loading both bytes of an address into a pair of registers. If you just need to load the high byte of an address, you can do something like this:

```
10          brk
11          nop
12          lda   #label   ;load low byte
13          lda   #/label  ;& replace w/hi byte
```

You must specify the same label in lines 12 and 13 so that the relocator can properly adjust for code that isn't on a page boundary. To get the low byte only, try this:

```
10          brk
11          nop
12          lda   #label   ;load low byte
13          bit   0        ;do almost nothing
```

The BIT 0 in line 13 will get adjusted, but notice that it's a page zero instruction, not an immediate instruction. The relocator doesn't care and will adjust its operand anyway; thus, the BIT instruction will operate on some arbitrary location on page zero, changing your Negative and Zero status flags but leaving the actual value in the accumulator unchanged.

Really Breaking

If you really want to execute a BRK instruction, just include two of them in a row, like this:

```
50      brk         ;come to a screeching halt
51      brk
```

This will bring your program to a stop in the Monitor with a program counter and register display. You'll notice that the program counter has been adjusted to point to the first BRK instruction, rather than two bytes past that address as it usually does when a BRK is executed.

De-installing the Run-Time Relocator

Sometime before your program ends, you'll probably want to disconnect the run-time relocator from the BRK vector and reconnect the standard Apple BRK routine (or whatever BRK-handler was active when the run-time relocator began execution). A relocatable JSR to RTR_DINS will do the trick for you:

```
60      brk                     ;de-install RTR
61      jsr   RTR_DINS
```

Remember, you won't be able to use the run-time relocator after de-installing it, so be sure that you really are getting ready to exit your program before calling RTR_DINS. RTR_DINS does not change any registers.

How It Works

The run-time relocator uses five global labels: RTR, RTR_OLDB, RTR_PROC, RTR_DINS, and RTR_CONT. All the rest are local labels, and in several places I used hex addresses instead of EQU'd labels. The goal was to be able to PUT the code without having to remember what labels I had "used up" in the relocator.

The run-time relocator uses some tricky code, but really isn't that complex. The main keys to understanding what is going on are the ORG $800 statement (line 13), which specifies where the program is "set up" internally to run, and some knowledge of how the BRK handler works. When a BRK instruction is encountered, the 6502 jumps to the NMI handler in the Monitor. The Monitor saves the program counter and all registers, then decides whether the interrupt was caused by a BRK instruction or an actual NMI, and passes control to the appropriate handler through a page 3 vector. We can take advantage of the saved information in our BRK handler, and we can use any registers we like as long as we call the Monitor routine to reload the registers before we exit. With this information in mind, let's look at the program's main "chunks" of code.

Lines 19 through 54 are what gets executed when you run your program. First, all registers and the current contents of the BRK vector are saved on the stack. Line 29 calls a known RTS instruction in the ROM, then lines 30-33 adjust the stack pointer so the return address can be retrieved with PLAs. The return address is adjusted to point to our BRK handler, then stored into the BRK vector (lines 34-40). The old contents of the BRK vector are then pulled off the stack and stored into RTR_OLDB, using the run-time relocator itself to do the dirty work of relocating the STA instructions. Finally, in lines 47-52, the registers are restored, and the main program (which begins at RTR_CONT) is entered via a relocatable JMP instruction, again using the run-time

relocator.

Lines 64-74 are the main BRK handler routine. The very first thing the relocator does is to back up the program counter ($3A-$3B) by two bytes, because the BRK instruction always generates a return address two bytes past the BRK. In lines 70-74 I handle the two special cases, effective addressing and true BRK. If neither is found, the instruction is assumed to be three bytes in length.

Lines 78-101 handle effective addressing. You'll need to keep a close eye on the Y register to understand this routine. Treating the operands of the two immediate-mode instructions as an address, this routine subtracts out the assembly-time address of the BRK routine, then adds in the run-time address of the BRK handler as stored in the BRK vector. The result is stored back into the actual program code, and the routine exits through line 104. No checking is done to ensure that there really are two immediate-mode instructions present; the relocator assumes you know what you're doing.

Line 105 handles a true BRK (two BRK instructions in a row) by JMPing to $FA59, the Monitor ROM's default BRK handler. This ROM routine displays the registers and program counter and drops into the Monitor.

The handler for the three-byte instructions (lines 109-126) works in essentially the same way as the effective addressing handler at lines 78-101. The Y register is incremented and decremented to point at slightly different bytes, but the function is the same. Once again no opcode checking is done; it's up to you to make sure you include a three byte instruction after a BRK.

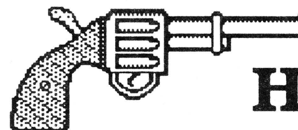
The exit routine, used by both the effective address handler and the absolute address handler, is at line 128. This short routine stores a NOP over the BRK instruction which called the relocator, restores all registers with a call to $FF3F, and JMPs back into the main program. Notice that the BRK handler (line 81) does NOT unpatch the BRK. If it did, the two BRKs (which mean a true BRK) would turn into one BRK, which, on subsequent executions, would cause the next instruction to be adjusted!

CONCLUSION

The run-time relocator is a slick way to write code that can be instantly run at any location without having to mess around with adjusting it. It's easy to use, and flexible too.

Naturally, it's not perfect. If you move the program to another memory location after it's been executed once, it won't work because all the BRKs have been patched out. Your code does end up slightly larger because of the BRK interpreter and the needed BRKs. You have to be careful if you're fond of self-modifying code because the BRK handler does some modifying of its own. If the user presses Reset, an instruction may wind up half-relocated. Et cetera.

It is, however, very well suited to programs that will be loaded into memory at an arbitrary address, executed once, and then abandoned, such as short BASIC.SYSTEM utility programs. Besides which, run-time relocation is a neat thing you can do with your Apple, and it's fun and instructive to boot. Isn't that enough of a reason to play around with it?



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

David Ely. 4567 W. 159th St. Lawndale, CA 90260. 213-371-4350 eves. or leave message. GEnie: [DDELY], AOL: "DaveEly". Experienced in 8 and 16 bit assembly, C, Forth and BASIC. Available for hourly or flat fee contract work on all Apple II platforms (IIgs preferred). Have experience in writing desktop and classical applications in 8 or 16 bit environments, hardware and firmware interfacing, patching and program maintenance. Will work individually or as a part if a group.

Jeff Holcomb, 18250 Marsh Ln, #515, Dallas, Tx 75287. (214) 306-0710, leave message. GEnie: [Applied.Eng], AOL: "AE Jeff". I am

looking for part-time work in my spare time. I prefer 16-bit programs but I am familiar with 8-bit. Strengths are GS/OS, desktop applications, and sound programming. I have also worked with hardware/firmware, desk accessories, CDevs, and inits.

Tom Hoover, Rt 1 Box 362, Lorena, TX, 76655, 817-752-9731 (day), 817-666-7605 (night). GEnie: Tom-Hoover; AOL: THoover; Pro-Beagle, Pro-APA, or Pro-Carolina: thoover. Interests/strengths are 8-bit utility programs, including TimeOut(tm) applications, written in assembly language. Looking for "part-time" work only, to be done in my spare time.

Jay Jennings, 14-9125 Robinson #2A, Overland Park, KS, 66212. (913) 642-5396 late evenings or early mornings. GEnie: [A2.JAY] or [PUNKWARE]. Apple IIgs assembly language programmer. Looking for short term projects, typically 2-4 weeks. Could be convinced to do longer projects in some cases. Familiar with console, modem, and network programming, desk accessories, programming utilities, data bases, etc. GS/OS only. No DOS 3.3 and no 8-bit (unless the money is extremely good and there's a company car involved).

Jim Lazar, 1109 Niesen Road, Port Washington, WI 53074, 414-284-4838 nights, 414-781-6700 days. AOL: "WinkieJim", GEnie: [WINKIEJIM]. Strengths include: GS/OS and ProDOS 8 work, desk top applications, CDAs, NDAs, INITs. Prefer working in 6502 or 65816 Assembly. Have experience with large and small programs, utilities, games, disk copy routines and writing documentation. Nibble, inCider and Call-A.P.P.L.E. have published my work. Prefer 16-bit, but will do 8-bit work. Type of work depends on the situation, would consider full-time for career move/benefits, otherwise 25 hrs/month (flexible).

Stephen P. Lepisto, 12907 Strathern St., N. Hollywood, CA 91605, 818-503-2939. GEnie: S.LEPISTO. Available for full-time and part-time contract work (flat rate or royalties). Experienced in 6502 to 65816 assembly, BASIC and C. Can work in these or quickly learn new languages and hardware (some experience with UNIX, MS-DOS, 8086 assembly). Experience in games, utilities, educational, applications. Lots of experience in porting programs to Apples. Programmed Hacker II (64k Apple II), Labyrinth (128k Apple), Firepower GS and others. Can also write technical articles.

Chris McKinsey, 3401 Alder Drive, Tacoma, WA, 98439, 206-588-7985, GEnie: C.MCKINSEY. Experience in programming 16-bit (65c816) games. Strengths include complex super hi-res animation, sound work (digitized and sequenced), and firmware. Looking for new IIgs game to develop or the porting of games from other computers to the IIgs.

We'll be running those folks with last names starting with M-Z next month!

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!

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

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

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

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**

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

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

Your satisfaction is our guarantee!

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.

http://apple2scans.net