The Sourceror's Apprentice

The Assembly Language Journal of Merlin Programmers Vol. 1 No. 2 February, 1989

Philosophy 101, Screw Ups, and AlertWindow

Tom Weishaar, editor of *A2-Central/Open-Apple* once called *Apple Viewpoints* a "...miserable little weekly newsletter for developers..." I tend to be a little easier on the poor shmucks (Hey, MONTHLY newsletters are tough to publish. I can't begin to imagine the hassles with a weekly. I picture their staff wandering the halls at Apple's corporate HQ buttonholing anyone with a pulse to ask, "Will you PLEASE write a guest column this week?" I'm waiting for the views of a janitor to slip into print.) At any rate, *Apple Viewpoints* is indeed pretty slim pickins for Apple II developers.

This last week, though (February 6th), Jean-Louis Gassee did their weekly guest column and entitled it, "Algorithms Always Break Down". Gassee has always struck me as a fairly deep thinker, so I dug right in. Even though he mentioned the Mac (I think his target was the Mac OS) and not the Apple II, his main point still reflects a uniquely Apple II worldview:

"*This is my point:* Sometimes, just sometimes, beauty - and ultimately cost-effectiveness - can come from *restraint.*"

I would like to add that I would rather have a program (or even a computer) which does a few things exceedingly well than one which does many things without distinction. There are occassions on which I like the IIc+ better than the GS, for example.

Just a little philosophizing for you to do with what you will.

I tend to wax a little philosophical when I booger something up. I've been doing a lot of waxing the last few days (and boy, do I have even MORE respect for Bob Sander-Cederlof and Tom Weishaar). The Generic Startup code we printed in Volume 0, Number 0 was, as Don Lancaster might put it, "fatally flawed". I can think of different ways to say it.

At any rate, we got straightened out by the completely revised Apple II Technical Note suite recently released by Developer Technical Support. Kudos to Jim Mensch, Matt Deatherage, and the rest of the gang for finishing a monumental task. Wish I'd had them three months ago, but I know quality takes time. T'ain't perfect (I'll show you what I mean later), but the new notes still bring together a lot of stray information.

Anyway, back to Generic Startup. The first and most critical bug lives at line 332 of our old code, where we had

```
332             ~DisposeHandle ProgID
```

I am afraid we understood the issue here, but printed erroneous code anyway. One of our readers thought we should have used `DisposeAll`, but that was not really the error. You MAY use `DisposeAll` if you have garnered memory space using an *auxiliary ID*, but you certainly would NOT pass it your program ID. If you did that, the very space in which you were operating would be deallocated. You would be trashing your own program, bombing your own buildings, blowing up your... I hope you get the picture. Under normal circumstances, `DisposeAll` requires an auxilliary ID. It then deallocates all of the memory attached to that particular ID (usually direct page space and data blocks).

In our case, we allocated additional memory using our **unmodified user ID** (our intention was to keep the code as simple as possible). In such instances, you must use `DisposeHandle` for each and every handle you've acquired. What you've messed up, ya gotta clean up. `DisposeHandle` requires, of

course, a handle to dispose of.

Line 332 should be changed, then, so that we are passing our handle:

```
332             ~DisposeHandle DPptr
```

The only memory we allocated in our startup routine was all grabbed with the handle DPptr (beginning at line 98 on p.8). Our bug, then, was the result of passing an incorrect parameter.

The remainder of the problems or changes in Generic Startup had to do with error checking (we didn't always need to when we did) and the manner with which we dealt with ROM tools.

The ToolBox Reference manual states that, "...you don't need to keep track of where a particular function is or even whether it is in ROM or RAM. A tool set called the Tool Locator...takes care of the necessary bookkeeping functions." (Volume 1, p. 1-1)

In spite of that pronouncement, every example startup routine I've looked at (including our old one) has treated the ROM tools as a separate case. They are all started without being loaded by a `LoadTools` call. Although this works, of course, I think there is a better way.

First of all, as we mentioned in Volume 0, Number 0 there is the often overlooked matter of minimum versions for NDA support. Even some of the ROM tools have minimum versions required for fully confident support of NDAs.

In our old code, we approached the issue by doing version calls for each ROM tool. A better approach, in my view, is to start the Tool Locator and then use `LoadTools` to pull in ALL of the tools, even the ROM based tools. This at least saves us some code in that `LoadTools` checks version numbers (based on the contents of the tool table data). And it seems to be the process Apple's toolbox designers envisioned. Heck, it might even extend the compatibility of your programs. Who knows what will or won't be in ROM next year?

In this month's Generic Start/ShutDown code, I started up two more tools than last time: the Font Manager and Quick Draw Auxiliary. I did this to support two of the more useful toolbox calls - `LEText2` and `AlertWindow`. `LEText2` permits the easy display of text within a box. The text is automatically justified and wrapped, too, so it is pretty useful.

`AlertWindow` greatly simplifies the construction of alert boxes. It is worth some attention - certified developers are really the only folks that know much about it since they receive updated information with each of the new system disks. Those of you trying to learn GS programming via the popular press are going to be nearly a year behind the times. There is not a book available today that documents the new `AlertWindow` call (I know, because I have them ALL - I think). The call became available on System Disk 3.2 which was released in August. That just isn't enough time to include it in a new book. That's where we in the newsletter industry come in handy. We can provide immediate gratification, especially now that Apple is going to charge at least $600 for the gratification of being a Certified Developer. But that's a different column. Today I'll expound on the `AlertWindow` call immediately following the revised Generic Start/ShutDown code below.

Note that there is a bit of a trick involved in getting the Font Manager started. The Font Manager searches for the FONTS subdirectory on the boot disk; it must find the fonts, obviously. But such a search can generate a VOLUME NOT FOUND error if that disk is not in a drive. Such a situation shouldn't crash the system, of course, so the program must screen error $45 when starting the Font Manager.

I also wanted to mention Mohawk Man's `Check4Error` subroutine (similar to APW's ERRORDEATH routine). This routine is intended primarily for debugging puposes. If your program crashes, the routine will print the error number and the location of the error. That is why MM loaded the X register with a number before calling it; the number is a sort of ID to which you can refer in case of a crash. In

your own polished and debugged software, such an error handler should be replaced by something a little more elegant - like maybe the Mac's bomb (snicker, snicker).

```
 1 ********************************
 2 *                              *
 3 *  Generic Start v 2.0         *
 4 *  By Mohawk Man, Ross Lambert,*
 5 *  and Eric Mueller            *
 6 *  11/15/88                    *
 7 *                              *
 8 *  Revised by Ross W. Lambert  *
 9 *  2/15/89                     *
10 *                              *
11 *  This program is public      *
12 *  domain...                   *
13 *                              *
14 ********************************
15
16
17
18 * Stuff for Merlin
19
20
21          mx    %00                ;full 16-bit mode
22          rel                      ;generate relocatable output
23          dsk generic.start.l      ;name of output file (link file)
24          use gen.strt.macs        ;run MACGEN to generate this file
25
26          lst   off
27
28 *
29 * Our equates:
30 *
31
32
33 Attr     =     %11000000_00000101 ;attributes for memory allocation
34                                    ;(lockd, fixd, pg alignd, & fixed bk)
35 DPPtr    =     0                  ;our Direct Page pointer location
36 TotalDP  =     $800               ;total number of direct pages needed
37 scrnsize =     0                  ;allows default ($80 for 640 mode)
38
39
40 * These establish full screen environment for QD (320 mode)
41
42 maxx     =     320                      ;maximum x value
43 minx     =     0                        ;minimum x value
44 maxy     =     200                      ;maximum y value
45 miny     =     0                        ;minimum y value
46
47
48
49
50 *
51 * One macro for fun (maybe move to a macro library):
52 *
53
54
55 IncDP    mac
```

```
56          lda   DPAddr
57          clc
58          adc   #]1             ;add number of pages in variable
59          sta   DPAddr          ;]1 to our current dircect pg ptr.
60          EOM
61
62
63
64 *
65 *   startup the needed tools
66 *
67
68 StartUp  ent                   ;this label now global if any other
69                                ;module  needs it.
70          phk
71          plb                   ;make the program bank = data bank
72
73          _TLStartUp            ;start the Tool Locator via macro
74
75
76 *
77 * load all tools - let Tool Locator worry about ROM vs. RAM
78 *
79 *
80
81
82 LoadTools ~LoadTools #ToolTable
83          bcc   StartTools       ;if the tools loaded okay, start them
84
85          cmp   #$45             ;was error VOLUME NOT FOUND?
86          bne   AbortStart       ;if not, we be dead
87          jsr   GetBootVol       ;get the boot volume for the tools
88          cmp   #1               ;'Okay' picked?
89          beq   LoadTools        ;if so, try to load the tools again
90
91 AbortStart
92          ~MoveTo #50;#90
93          ~DrawString #MSGb      ;tell user we crashed
94
95          jmp   EarlyShutDown    ;we're outta here
96
97
98 * Start up tool sets.  NOTE: If there is not an error handling routine
99 * for a particular tool set, it is because the start up call for that
100 * tool set does not return any errors.
101
102
103 StartTools
104
105         _ADBStartUp            ;recommended by Apple, Inc.
106
107         _IMStartUp
108
109 :1      ~MMStartUp             ;start the Memory manager
110         ldx   #1
111         jsr   Check4Error
112
113         pla                    ;retrieve our program ID
114         sta   ProgID
115
116                                ;next, get direct page memory for tools
117
118 :2      ~NewHandle #TotalDP;ProgID;#Attr;#0
```

```
119
120           ldx   #2
121           jsr   Check4Error
122
123           PullLong DPptr          ;get handle and store in our dir. pg
124           LDA   [DPptr]           ;long indirect load gets new dp address
125                                   ;for tools
126           STA   DPAddr            ;store it
127
128           _MTStartUp              ;start Miscellaneous Tools
129                                   ;recommended by Apple to insure
130                                   ;future compatibility
131
132
133
134 :3       ~QDStartUp DPAddr;#scrnsize;#0;ProgID
135           BCC   :4
136           CMP   #$0401            ;already started?
137           BEQ   :4                ;if so, ignore it
138
139           ldx   #3                ;go find out what else is wrong...
140           jsr   Check4Error
141
142 :4       IncDP $300              ;QD uses three pages
143
144           _QDAuxStartUp           ;start Quick Draw Auxiliary
145                                   ;required for LETextBox2
146                                   ;LongStatText2 (both are useful)
147
148           ~EMStartUp DPAddr;#20;#minx;#maxx;#miny;#maxy;ProgID
149           BCC   cont1
150           CMP   #$0601                    ;ignore already started error
151           BEQ   cont1
152
153           ldx   #4
154           jsr   Check4Error
155
156 cont1     IncDP $100                      ;event manager used one page
157
158           ~WindStartUp ProgID             ;start the Window Manager
159           ldx   #5
160           jsr   Check4Error
161
162           ~RefreshDeskTop #0              ;draw screen/ 0 = whole scrn
163
164           ~CtlStartUp ProgID;DPAddr       ;start the Control Manager
165           ldx   #6
166           jsr   Check4Error
167
168           IncDP $100                      ;Control Mgr. needs direct page
169
170
171           ~MenuStartUp ProgID;DPAddr      ;start the Menu Manager
172
173           IncDP $100
174
175           ~LEStartUp ProgID;DPAddr        ;start Up line edit tools
176           ldx   #7
177           jsr   Check4Error
178
179           IncDP $100                      ;uses a direct page
180
181           ~DialogStartUp ProgID           ;start the Dialog manager
```

```
182
183           _ScrapStartUp                  ;scrap mgr for NDA support
184
185           _DeskStartUp                   ;start the Desk Manager
186
187           _ShowCursor                    ;show arrow in case needed for
188                                          ;TLMountBootVol dialog
189
190 doFontMgr
191           ~FMStartup ProgID;DPAddr       ;necessary for "AlertWindow"
192           BCC   AOK                      ;AOK - continue
193           CMP   #$45                     ;VOLUME NOT FOUND (needs FONTS
194                                          ;subdirectory on boot volume)
195           BNE   mo_err
196           JSR   GetBootVol               ;prompt for boot volume
197           CMP   #1                       ;OK?
198           BEQ   doFontMgr
199           JMP   ShutDown                 ;user wants to cancel, so quit
200
201 mo_err    ldx   #8
202           jsr   Check4Error
203
204 AOK       IncDP $100                     ;uses one direct page
205
206 *
207 * StartUp procedure is finished.
208 *
209 *   Your program would continue here.
210 *   This trivial example is mostly to prove
211 *   that it all works!
212 *
213
214
215           ~SetBackColor #13              ;change background of text block
216           ~MoveTo #20;#90                ;X & Y coordinates
217           ~DrawString #MSGa              ;print 2nd message (press a key)
218
219 RDKEY     LDAL  $C000                    ;read keyboard for a keypress
220           AND   #$00FF                   ;clear high word
221           CMP   #$0080                   ;keypress?
222           BCC   RDKEY                    ;if not, loop back
223           STAL  $C010                    ;clear strobe
224
225           JMP   ShutDown                 ;if a key is pressed, shutdown
226
227
228
229 MSGa      STR   'Press any key for shutdown...'
230 MSGb      STR   'Premature shutdown: LoadTools failure...'
231
232
233 *
234 *   Check For Errors After
235 *     Making A Tool Call
236 *
237
238 Check4Error bcs YepError                 ;carry set if there was an error
239           rts                            ;otherwise, get outta here
240
241 YepError phx           ;save the error code on stack for _SysFailMgr
242           sta   ErrCode                  ;save our error number for a sec
243           ~HexIt ErrCode                 ;change code to a hex number
244           PullLong ErrCode               ;save and store error code
```

```
245          PushLong #DeathMsg          ;use customized death message
246          _SysFailMgr                 ;oh! we're dead!
247
248 DeathMsg
249          dfb   #EndDMsg-StartDMsg
250 StartDMsg
251          asc   'Error $'
252
253
254 ErrCode  ds    4
255          asc   ' at location $'
256 EndDMsg
257
258
259 *
260 * Get the boot volume
261 *
262
263
264 GetBootVol
265
266          _Get_Boot_Vol GBVParms       ;get the name of the boot volume
267
268          ~TLMountVolume #25;#50;#Prompt1;#VolName;#OKMsg;#CancelMsg
269                                       ;ask the user for the boot volume
270
271          pla                          ;retrieve the result
272          rts                          ;and get outta here
273
274 GBVParms adrl  VolName        ;pointer to space for the volume name
275
276 Prompt1  str   'Insert the volume: '
277 VolName  ds    18                    ;space for volume name
278
279 OKMsg    str   'OK'
280 CancelMsg str  'Cancel'
281
282
283
284 *
285 *  A Generic ShutDown Routine
286 *
287
288 ShutDown ent         ;global label so other modules can die here.:)
289
290          _DeskShutDown             ;kill this 1st so NDAs are history
291          _FMShutDown
292          _ScrapShutDown
293          _DialogShutDown
294          _LEShutDown
295          _MenuShutDown
296          _WindShutDown             ;Window Mgr must be shut down before
297          _CtlShutDown              ;the Control Mgr
298          _EMShutDown
299          _QDAuxShutDown
300          _QDShutDown
301          _MTShutDown
302          _IMShutDown
303          _ADBShutDown
304
305          ~DisposeHandle DPptr      ;release handle acquired by ProgID
306          ~MMShutDown ProgID
307
```

```
EarlyShutDown                       ;Tool Locator only tool started if
          _TLShutDown               ;EarlyShutDown called

:1        _Quit QuitParms
          bra   :1                  ;keep quitting if GS/OS is busy

QuitParms adrl $0
          ds    2

*
* some data for the modules
*

ProgID    ent                 ;make it global
          ds    2             ;space for the program ID
DPAddr    ent                 ;also global
          ds    4             ;pointer to the direct pages


*******************************************************
*
* Meaning of the double numbers in the table:
*
*  The first number is the tool number and
*   the second number is the minimum version
*   needed for our application.
*
* (editor: you may need to change these if your
* application requires certain functions/bug fixes
*  only available in updated versions of the tools.)
*
*  The version number is hexedecimalized, i.e. v 1.2
*  is turned into $0102, v 1.3 = $0103, etc.
*  A zero means we'll take anything.

*******************************************************


ToolTable dw    15          ;number of tools to load


          dw    2,$0102     ;Memory Manager
          dw    3,$0102     ;Miscellaneous Tools
          dw    4,$0102     ;Quick Draw II
          dw    5,0         ;Desk Manager
          dw    6,0         ;Event Manager
          dw    9,0         ;ADB Tools
          dw    10,0        ;Integer Math tools
          dw    14,$0103    ;Window Manager
          dw    15,$0103    ;Menu Manager
          dw    16,$0103    ;Control Manager
          dw    18,0        ;Quick Draw Auxilliary
          dw    20,0        ;LineEdit tools
          dw    21,$0101    ;Dialog Mgr (v 1.1 rqd by Std. File, Print
                            ;Mgr, etc.)
          dw    22,0        ;Scrap Manager
          dw    27,0        ;Font Manager
```

I promised some information regarding the AlertWindow toolbox call... AlertWindow is really a time and code saver since it greatly simplifies the constuction of alert boxes. The following information is adapted from Apple IIgs Tech Note 48 and the Toolbox Reference Update (beta version).

There are two things to understand about the call. The first is that it makes use of some "standard" alert box and icon definitions. You can pick which one of the standards you want to use or tell the tool your own parameters. It is a remarkably flexible call.

The standard box sizes look like this:

Character	Height 320	Width 320	Height 640	Width 640
1	46	152	46	200
2	62	176	54	228
3	62	252	62	300
4	90	252	72	352
5	54	252	46	400
6	62	300	54	452
7	80	300	62	500
8	108	300	72	552
9	134	300	80	600
0	(Character followed by 4 integers - Height, Width, Xpos, & Ypos)			

The Character column is the actual ASCII character you pass to the tool to indicate your selection. More on that in a moment.

AlertWindow also lets you choose from several standard icons. The icon number can be 0-9 where :

```
0 = no icon
1 = custom icon, followed by:
    LONG -- pointer to image data
    WORD -- number of bytes image data is wide
    WORD -- number of scan lines image data is high
2 = Stop icon
3 = Note icon
4 = Caution icon
5 = Disk icon
6 = Disk swap icon
7 - 9 Reserved - do not use
```

This first step in the building process is to set up what is called an "alert string". A simple one could look like this:

```
ASC   '13/Simple Alert Box/Button 1',00
```

Notice that, like almost all toolbox related strings, the string data must be positive ASCII (denoted by the single quotation mark in Merlin). You can also see that the first two characters are numbers. The first number represents the chosen standard box size, while the second represents the chosen icon. Immediately following those, you must insert a separator character. It can be any character not actually in the string itself, but it is best to use something odd like the backslash or the percent sign. DTS

recommends the backslash for purposes of standardization.

Next comes the text of the alert message. It is followed by the separator and the text of the first button. Watch out here, though, because you are limited to three buttons. If you need more than that, ya gotta do it the old fashioned way.

The string terminates with a zero, which is the flag for the tool.

To actually call the tool, you must pass the type of string data you're sending (does it have a leading length byte or terminate with a CR or a zero?), the address of the substitution array (more on that in a minute), and the address of the alert string.

But first, a macro.

I cannot imagine programming the GS without macros. So here is one that will make `AlertWindow` do its thing mo' betta. This macro should be appended to the file WINDOW.MACS.S in the TOOL.MACROS subdirectory of your Merlin disk.

```
~AlertWindow    MAC
                PHA                 ;push dummy return space onto stack
                PHWL ]1;]2          ;push word length string type, then
                                    ;long address of substitution array
                PHL  ]3             ;long address of alert string
_AlertWindow    tool $590E          ;use tool macros to do call
                <<<                 ;that's all folks
```

Now, to call `AlertWindow`, just do this:

```
~AlertWindow #stringtype;#longaddrSubarray;#longaddrAlertstring
```

The string type we keep mentioning is defined as zero for a C string (which terminate with a zero or CR) or a one for a Pascal string (which has a leading length byte). If you prefer Pascal strings, you would use the STR operand in Merlin.

Not so incidentally, after calling `AlertWindow`, you must do a PLA to pull the number of the button pressed off the stack.

The only topic we've not covered yet is the substitution array. As you probably figured out, if you're not using this feature, just pass a zero to the function. If you are, pass its long address.

So what it is, man? Well, okay. The substitution array is a mechanism whereby you can arbitrarily insert blocks of text into your alert string. For example, take a look at the following:

```
ASC   '42/This *0 *1 text that *2 array./*3/^Button 2',00
```

The asterisks followed by a number in the range 0-9 tells `AlertWindow` to go look for the appropriate string data from the substitution array and stick it into the alert string at that very spot.

The substituion array itself looks like this:

```
SubArray
          ADRL   Sub0
          ADRL   Sub1
          ADRL   Sub2
          ADRL   Sub3

Sub0      ASC    'is',00
Sub1      ASC    'substitution',00
Sub2      ASC    'we transferred from our',00
Sub3      ASC    'Sub Text',00
```

At the long address we pass to the toolbox (SubArray), we stash the long address of each of the subsitution strings we want to use in our alert string. `AlertWindow` then just goes and looks them up and plops them in. These strings MUST be of the same type (C or Pascal) as the main alert string, by the way.

If you happen to peruse the actual tech note from Apple, do not use the example for the alert string with substitutions that they offered. The icon number they put into their alert string was for a custom icon, which makes the function look for icon definition data. This gums up the works if you don't have a custom icon. I almost lost my mind trying to figure out what I was doing wrong. I told you that the tech notes were not perfect!

In any event, there are a couple of final tidbits which can be of use.

First, if you want to make a button the default button, preceed its text with a caret (^).

Second, `AlertWindow` has some standard text options to choose from as well as standard boxes and icons.

```
#0 = OK
#1 = Cancel
#2 = Yes
#3 = No
#4 = Try Again
#5 = Quit
#6 = Continue
```

Do not use 7-9.

As is obvious, preceed the number of the standard text item you'd like with a pound sign (#).

If you'd like a quick and dirty demo, insert the following lines in place of the keyboard scanning routine at lines 206-228 of the Generic Startup routine. Your new little "application" will start all of the tools, cycle you through several different types of alert boxes, and then cleanly shut you down.

Have fun.

```
1
2           ~AlertWindow #0;#0;#AlertString1
3            PLA                ;ignoring button pressed for now
4
5           ~AlertWindow #0;#0;#AlertString2
```

```
 6           PLA                     ;ignoring button pressed
 7
 8           ~AlertWindow #0;#0;#AlertString3
 9           PLA                     ;ingore it for now
10
11           ~AlertWindow #0;#SubArray;#AlertString4
12
13           JMP    ShutDown
14
15
16 AlertString1
17           ASC    '13/Simple Alert Box/Button 1',00
18
19 AlertString2
20           ASC    '26/A Disk-Swap Alert Box/^Button 1/Button 2',00
21
22 AlertString3
23           ASC    '34/Standard button text/^#0/#6',00
24
25 AlertString4
26           ASC    '42/This *0 *1 text that *2 array./*3/^Button 2',00
27
28 SubArray
29           ADRL   Sub0
30           ADRL   Sub1
31           ADRL   Sub2
32           ADRL   Sub3
33
34 Sub0      ASC    'is',00
35 Sub1      ASC    'substitution',00
36 Sub2      ASC    'we transferred from our',00
37 Sub3      ASC    'Sub Text',00
```

Relocation Without Dislocation

By Karl Bunker
321 S. Huntington Ave.
Boston, MA 02130

Editor: Karl Bunker is the author of the popular public domain program, "DOGPAW". DOGPAW reads, displays, formats, and prints either AppleWorks files or text files. It is a marvelous tool for viewing and printing on-disk documentation.

There are any number of reasons why you might want to write an 8-bit program which is relocatable, that is, which can run at any available memory address. If a program uses a call to GETBUFR to allocate a buffer from BASIC.SYSTEM, and then relocates part of itself into this buffer, the address of its new location will depend on whether or not your program was the first to make a call to GETBUFR. Or if you are writing a routine that will be attached to the end of an Applesoft program, its address will depend on the size of the BASIC program. You may even need to write a program which can simply be BRUN at any (legal) address, perhaps so that it can co-reside in memory with some other program(s). In situations such as these, your code will have to be relocatable, and while this presents some interesting problems, they are by no means insurmountable.

The simplest way to make 8-bit code relocatable, of course, is to avoid absolute addressing when the operand refers to an address within the program. Thus, instead of:

```
          JMP       DEST
```

Relative addressing could be used:

```
          CLV
          BVC       DEST
```

Editor: you could also use BRA DEST which is available on the 65C02 and later microprocessors.

However, relative addressing is limited to jumps of 127 bytes or less - not very far. This limitation can be overcome, at the cost of somewhat greater complexity, by using one or more relative addressing "bounces", like so:

```
          CLV
          BVC       BOUNCE
                              ;Up to 127 bytes
                              ;of code here.
BOUNCE    BVC       DEST.
                              ;Another 127 bytes
                              ;(or less) here.
DEST                          ;Final destination
```

Of course, the intervening code will have to hop over the BVC at BOUNCE. The difficulty of using this method goes up sharply as the number and length of jumps in the program increases. Also, there is no relative addressing equivalent to a JSR/RTS, or to other absolute addressing operations. For example, a data byte that is inside the relocated code can't be accessed with an LDA or STA.

To make 8-bit code relocatable while still retaining the full use of all absolute-addressed instructions requires self modifying code. That is, the relocatable portion of the program is written as usual, with a fixed ORG. But when the program is run, a routine within the program modifies the code, changing all of the address operands that need adjusting for the program's new location.

As was indicated above, there are basically two types of situations that call for relocatable code. First, a program may be BRUN at some fixed address, and then relocate a portion of itself to a new address which is unknown at the time the code is being written. Second, the whole program may simply be "dropped" into some arbitrary address which is unknown at the time of its writing. The second of these is a little trickier to handle, so let's start by looking at the first.

In this type of situation, the program can be divided into a "relocator module" and the "relocated module". The relocator module will first determine what address the relocated module will be going to, then adjust the address operands of the relocated module as needed, and finally move it to its new location. If there are only a few internal JMP's and JSR's in the relocated module, they can simply be given labels, such as JUMP_1, JUMP_2, etc., and then, before relocating the code, the addresses can be modified, like so:

```
          CLC
          LDA       JUMP_1+2  ;High byte of jump address
          ADC       DIFRNC    ;Add what's needed for new
          STA       JUMP_1+2  ;location, put it back.
;
REL_STRT                      ;Module to be relocated starts here
JUMP_1    JSR       WHEREVER
                              ;Etcetera
```

A few things are "assumed" in this segment of code: First, that the new location for the relocated module is higher in memory than its present address. Second, that the relocator module has determined the amount that has to be added to the high bytes of the addresses in the relocated module, and has stored this value in DIFRNC. Finally, this routine assumes that the old and new starting address have the same low byte. This will be the case, for example, if both are on a page boundary. GETBUFR allocates buffers by pages, so this a reasonable assumption.

Of course, this method can also be used to adjust absolute-addressed LDA's, etc. whose addresses are within the code being relocated. But since this method must handle each internal absolute addressing instruction separately, it becomes rather impractical with larger relocated modules.

Another way to adjust addresses before a module is relocated involves scanning through the code, using a Monitor routine to find those instructions which have 2 byte address operands, checking whether these addresses are internal to the code, and modifying them if they are. This Monitor routine, "INSDS2", is located at $F88E; the accumulator is loaded with an instruction byte, a JSR is done to this routine, and the address length of that instruction - 0, 1 or 2 - will be stored at $2F on page zero. If a 0 or 1 is found in $2F, the program simply skips over 0 or 1 bytes to get to the next instruction byte. If a 2 is found, the program will have to check whether the address that follows the instruction is within the code, and if it is, adjust it for its new location. Special handling is needed for data stashes; they can either be put at the end of the code, and the address-adjusting routine be set to stop scanning before it reaches them, or routines can be included in the address-adjuster to recognize and skip over data stashes. Here's a program segment that illustrates this code-modifying code; REL_STRT is the current, pre-relocation, starting address of the relocated module, and REL_END is the address of the end:

Editor: This example also assumes that you know the difference ,DFRNC, between the current address and the destination address. This is usually not difficult, especially for routines that load at a known address and relocate themselves to an address supplied by BASIC.SYSTEM's GETBUFR call, located at $BEF5. GETBUFR itself returns the highbyte of your new address in the accumulator. Since the least amount of space you can request from GETBUFR is one page, 256 bytes, the lowbyte of the address will always be zero.

One item to note in Karl's code, however, is that the part of the code to be relocated should be assembled on a page boundary. This allows you to ignore lowbytes.

```
          LDA       #<REL_STRT          ;Put current starting address
          STA       PTR                 ;of relocated module into some
          LDA       #>REL_STRT          ;0-page pointers.
          STA       PTR+1
ADJ_CODE  LDY       #0
          LDA       (PTR),Y             ;Get an instruction byte
          BEQ       ADJ_DONE            ;A 0 marks the end of the
                                        ;relocated module's code and
                                        ;the start of any data stashes.
          JSR       $F88E
          LDY       $2F                 ;Is this an absolute address
          CPY       #2                  ;instruction?
          BNE       NXT_INST            ;No, get next one
          LDA       (PTR),Y             ;Yes, get high byte of address
          CMP       #>REL_STRT          ;Is this address located inside
          BCC       NXT_INST            ;the relocated module's code?
          CMP       #>REL_END
          BEQ       FIX_ADR
          BCS       NXT_INST
FIX_ADR   CLC                           ;If so, fix it for its
          ADC       DIFRNC              ;new location,
          STA       (PTR),Y             ;and put it back.
```

```
NXT_INST INY                          ;Add 1 to what's in Y,
         TYA
         CLC
         ADC      PTR                 ;and add this to the pointers
         STA      PTR                 ;to skip ahead to the next
         LDA      #0                  ;instruction byte.
         ADC      PTR+1
         STA      PTR+1
         BNE      ADJ_CODE            ;(Always; high byte won't = 0)

ADJ_DONE                              ;Rest of relocator module's
                                      ;code here.
```

Now let's look at the other situation, where there is no relocator module; the whole program simply "finds itself" planted at some unknown address. The first thing this program will have to do is just that: find out where it is, so that it will know how to adjust its addresses. To do this, we can make use of the way in which the 65xx instruction set handles a JSR/RTS. When a JSR is encountered in a program, the "return address" is first saved on the stack, then the subroutine is jumped to. The return address, which is actually one byte less than the address of the instruction that follows the JSR's operand, tells the program counter where to go when it encounters the RTS that ends the subroutine. This return address - the home address our program - remains on the stack after the RTS is executed, and can be used to answer that vital question: "Where am I?"

Here's how it's done:

```
         PHP                  ;First save interrupt status & turn off
         SEI                  ;the interrupts so that an interrupt call
                              ;won't change what's in the stack.
         JSR      $FF58       ;This is the location of a known RTS in
                              ;the Monitor.
         TSX                  ;Stack pointer into X
         DEX                  ;Back up to get the low byte first
         SEC
         LDA      $100,X
         SBC      #2          ;Low byte of current address; put it
         STA      PTR         ;into 0-page pointer.
         INX                  ;Get high byte,
         LDA      $100,X
         SBC      #0
         STA      PTR+1       ;store it.
         PLP                  ;Restore interrupt status
```

The current, relocated address of the program - actually, the address of the JSR $FF58 instruction - is now loaded into some 0-page pointers, so the program is all set to scan through its own code and modify the addresses, just as was done above with the relocator module. In this situation, however, there will probably be no way to ensure that the program's current address has the same low byte as its ORG address, so both bytes of the absolute-address operands will have to be adjusted. Also, to keep things relatively simple, we should guarantee that the current address of the program is higher that its ORG address. This could be done by assembling the program with a "lowest possible" ORG; presumably $800.

More Relocation Without Dislocation...

Editor: You will have to decide yourself when to use each of the methods Karl has outlined. The primary considerations include the type and purpose of the software. A routine intended to "float" at the end of an Applesoft program will never be loaded at a known address, so it must use Karl's JSR trick to find out where it lives, modifying itself on the fly. ProDOS SYSTEM programs, on the other hand, are initially installed at $2000. Since that is smack in the middle of everything, you'll probably want to move. But a handy dandy relocating module can do the trick for you there. In any case, at least one of Karl's methods can be used in virtually any relocation situation.

Parting Thoughts

One of my friends told me that he really liked ***The Sourceror's Apprentice*** except that he wished that it was twice as long each month. I totally agree. I am afraid that I am a programmer first and a businessman second. That is why I have promised 12 pages monthly and have ended up delivering 16! I have got to be very careful, though, as you all probably realize. Generosity becomes self-defeating after a point. I have got to put food on the table or else I've gotta go hunt polar bear. And believe me, my daughter would get mighty hungry if her dad started living the life of a subsistence hunter.

Nevertheless, I am wondering if you assembly fanatics would rather have a $45-$50 newsletter (yearly) that runs 24 (8.5" X 11") pages or so each month? The quarterly diskette could remain the same price since the overhead for that would be constant. It won't happen anytime soon, but I'd be interested in hearing your thoughts. Drop me a note.

I've also had numerous questions regarding the name of my company, Ariel Publishing. The word Ariel is an old Hebrew word meaning, "Lion of God". As a born-again Christian, slipping some reminder of my faith into my company name is important to me (as is integrity and fairness). If you're ever dissatisfied with anything just throw that back in my face.

Kinda ironic, is it not, that both of the people (Bob S-C and I) who ventured into the assembly language newsletter arena were Christians? Well, we've got the Christians and the arena, where are the lions? (yuk, yuk)

By the way, Jerry Kindall's wonderful ***Applesoft Connection*** series will return next month. We've also got some joystick routines from Steven Lepisto that will knock your socks off.

Until then, then.

The Sourceror's Apprentice

Copyright (C) 1988 by Ross W. Lambert
and Ariel Publishing, Inc.
All Rights Reserved

All programs in THE APPRENTICE are in the public domain and may be freely copied and distributed. Apple User Groups and other important folks may reprint articles upon request. Just gimme a call at 907/624-3161 or drop me a line at the address below.

American prices in US dollars effective January 1, 1989: 1 yr..$28, 2 yrs..$52, Canada and Mexico add $5, all others add $10

Back issues are available at $3.00 each.

WARRANTY AND LIMITATION OF LIABILITY

I warrant that the information in THE APPRENTICE is correct and useful to somebody somewhere. Any subscriber may ask for a full refund of their last subscription payment at any time. MY LIABILITY FOR ERRORS AND OMISSIONS IS LIMITED TO THIS PUBLICATION'S PURCHASE PRICE. In no case shall I or my contributors be liable for any incidental or consequential damages, nor for ANY damages in excess of the fees paid by a subscriber.

Please direct all correspondence to:

Ariel Publishing, Inc.
P.O. Box 266
Unalakaleet, Alaska 99684 USA

THE APPRENTICE is a product of the United States of America.