May, 1990
Volume 1, No. 3

8/16

The Journal of Apple II Programming $3.50

Sweet 16 Again & "In Search of Bert"

This month in *8/16*:

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

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

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

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

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

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

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

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

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

Photo-Copyable

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

Sales Tax
Kansas residents 6%

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

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

(Cut & Paste Address Label)

Ship to: ______________________
____
____
____
Telephone #: ______________________

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

Credit Card or PO# | Expiration Date

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

Prices may Change without notice.

8/16

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

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

Introductory subscription prices in US dollars:

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

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

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

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.

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

The Publisher's Pen

by Ross W. Lambert

I recently came across the hippest, hottest, most happenin' computer bookstore I've ever seen. It's called The Computer Literacy Bookstore, and despite the dorky name (sorry Dan!) it has all the Apple II information ever printed since the dawn of time. The owner could tell me *off the top of his head* the status of an old, old Apple II book for which I've been searching for years. Although they are not totally dedicated to the II (you can get Mac stuff there also), the II is definitely where their heart is. According to owner Dan Doernberg they have something like 60,000 Apple II volumes in stock.

Now, I'm biased - they are the first folks to sell *8/16* over the counter. But *I* think that says something really terrific about them all by itself. I really doubt you'll be disappointed if you give them a call. Their number is **(408) 435-1118**. Tell 'em Mike Rochip sent you.

A Bert Kersey Sighting

Beloved Apple II pioneer Bert Kersey was recently sighted buying a six pack (of Coca-Cola) at a 7-11 outside of Barstow, CA. Rumor has it that he was in the company of Paul Lutus. Somebody get a camera and call *The Enquirer.*

What's the big deal about Bert Kersey, you ask? Why would I put his name on the cover of our April issue? I'll tell you why. Go read Guy Kawasaki's *The Macintosh Way.* It is a decent and humorous book. But it is the embodiment of the Macintosh (aka "Jobs") Arrogance. Ol' Guy writes like he and Steve invented the "Macintosh Way".

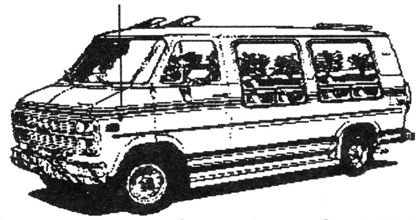
Sorry, Guy. Bert Kersey invented it. Only it is "The Apple II Way". Do you remember the first time you opened up a Beagle Bros product? I remember first of all that I was impressed with what a great deal it was for the money. I also remember that ol' Bert gave us mucho extras and goodies. I also remember laughing a lot. I also remember great service and personal responses from Bert himself. I have a handwritten note from him still in my files.

Best of all, Bert was doing these things when Guy Kawasaki was still having his mother wipe his nose. I still can't figure out why Bert's mother was wiping Guy's nose, though.

We here at Ariel are in search of Bert Kersey. We hope you are, too. == Ross ==

Sweet 16: A Blast From the Past

by Matt Neuberg



(Editor: If Matt is as successful resurrecting the classical (aka 'dead') languages he teaches as he is resurrecting Sweet 16, then it won't be long before Latin is back in vogue. BTW, Matt's stationery says "Lingua Optima, Lingua Mortua" on it, which I believe means "The best language is a dead language".)

The 8/16 paradox

Those who have replaced their chip with a 65802, or who have bought a GS, pay no attention. This is for the faithful few, still trying to live with the paradox of the 6502 and 65C02 machine: 8-bit processing, but 16-bit addressing. If you've ever written a machine language program of any size, you've faced the inconvenience of this paradox. Let's take an example.

Imagine we are faced with the following task. Suppose a textfile has been "packed" as in Apple Pascal: every time a run of 3 or more blanks occurs (up to 255), it has been replaced by a special character (say, $FF) followed by a byte containing the number of blanks. Our job is to decode ("unpack") the file, replacing each occurrence of "$FF N" with N blanks. Simple? Just wait.

Let two buffers be allocated in memory, starting at addresses FILEPK and FILEUNPK; suppose the packed file is already in place, starting at FILEPK, and a variable EOFPK points to the address after the last byte of the file. All we have to do is run through every byte from FILEPK up to EOFPK-1. If it isn't $FF, we'll just transfer it to the FILEUNPK buffer; if it is $FF, we'll look at the following byte and feed that number of blanks to the FILEUNPK buffer. We must also know the length of the resulting unpacked file when we are done, so we can save it.

Listing 2 shows a typical way to code it. It isn't fancy or tricky; but I don't mind telling you, it drove me crazy having to write it, and it isn't terribly easy to read, either.

The trouble is that this routine is all about manipulating addresses, and it takes two bytes to name and point to an address. See the large number of instructions taken up with handling 16-bit (2-byte) information one byte at a time ("lo-byte, hi-byte")? I've saved some space by relegating some of these operations to subroutines, but this doesn't make the program logic any clearer, and as I was writing the program, each time I realised I would need such a subroutine, I had to go back and change everything. And having to do everything twice, as it were, caused me to make stupid errors while coding, some of which I didn't catch until I tested the program. (In fact, there is still a bug in the program logic: can you find it?)

In my version, the whole task assembles to more than 110 bytes. This seems unnecessarily bulky. It might be possible to save a byte here and there, but this would require some clever coding, and more hard work. Isn't there a better way?

Enter Wozniak's dream machine

In the late '70's, when the great Steve Wozniak was designing Integer Basic for the Apple II, he encountered the 8/16-bit paradox, and devised an ingenious solution: Sweet16. I had seen references to Sweet16 - for example, MerlinPro assembles and disassembles Sweet16 code - but, like most people who got into micro-computers during the '80's, I had no idea what it was. *(My situation exactly - Editor)* Then one day, quite by accident, I found out.

Sweet16 is a 16-bit interpreter: i.e., it is 6502 code which simulates an imaginary processor, a processor capable of understanding and carrying out instructions which operate on 16-bit data. You can install this code in your Apple II, and when you have 16-bit operations to perform in an assembly-language program, you can have Sweet16 carry them out for you. Wozniak published the code for Sweet16 in the November 1977 issue of *BYTE* magazine. It is the basis for Listing 1 (discussed below).

Note: there are some errors in Woz's original article, so be careful if you hunt it up. But these have been eliminated in the following discussion.

As the Woz was quick to point out, Sweet16 is slow, probably 10 times slower than if the same tasks were performed by ordinary 6502 code. But once you've bought a computer, time costs you nothing; and besides, with today's accelerator boards and chips, you can make back almost half the extra time. On the other hand, the Sweet16 interpreter is ingeniously written in such a way that code written for it is extremely compact; this was the chief reason for Wozniak's inventing it, for in those days space inside the Apple II was at a premium. Moreover, Sweet16 gives the programmer another advantage, which Wozniak did not mention: it is extremely easy to code for. There are two reasons for this. First, you no longer have to keep track separately of each byte of your 16-bit data. Second, whenever you perform an indirect load or store through Sweet16, the 16-bit pseudo-register which points to that data is automatically incremented or decremented, making it very easy to operate on blocks of data, as in our example task.

The Sweet16 architecture

Sweet16 operates using 16 16-bit pseudo-registers. These occupy the space in the 0-page from $00 to $1F, and are designated R0-R15.

Some of these registers are special. The first register, R0, is the "accumulator". The last, R15, is used as the "program counter", telling the interpreter where to go to fetch an instruction from your program - either the next consecutive instruction, or the instruction to jump to after a branch. R14 is a "status register", used to point to the register in which the result of the last operation is stored, so that that result can be tested and, depending on its value, a branch can be performed. R14 also holds the "carry" bit, used for similar purposes. R13 is used by each CPR (compare) operation. R12 is used as a "stack pointer" when your Sweet16 code calls or returns from a subroutine (Sweet16 does not use the Page 1 stack), which means that if you use any subroutines in your Sweet16 code, you must have initialized R12 first, and you must not alter it during a subroutine.

R1-R11 are the registers free for your use, and will usually prove more than sufficient. There are no X- or Y-registers; but you won't need them, because indirect addressing is done automatically for you, and because, as mentioned above, Sweet16 itself always performs an increment or decrement of the register employed for indirect addressing.

The Sweet16 instruction set: register ops

In these descriptions, the shorthand "Rn" is used to designate a particular 16-bit register (usually R1-11). When assembling with MerlinPro, the "R" may be omitted.

Eight instructions operate directly with the various registers and the "accumulator". **Don't forget that these are 16-bit registers holding 16-bit values!**

`SET Rn,const` sets `Rn` to the value designated by `const`. With MerlinPro's assembler, this may be a previously EQUated label, a program line label, or an immediate value. The "#" symbol should not be employed, but if an immediate value is given, the "$" symbol may be. Remember, if you SET a register using a program line label, the resulting value will be the address of the line in memory, not whatever data appears in that line. (The comma is not required by MerlinPro; a space may be used instead, or nothing at all - e.g., `SET R3LABEL`).

`LD Rn` loads the "accumulator" with the 16-bit value in `Rn`
`ST Rn` stores the 16-bit value in the "accumulator" into `Rn`.

`INR Rn` increments `Rn`
`DCR Rn` decrements `Rn`. `Rn` can be the "accumulator" (`R0`).

ADD Rn adds the value in Rn to the value in the "accumulator", and leaves the result in the "accumulator".

SUB Rn subtracts the value in Rn from the value in the "accumulator", and leaves the result in the "accumulator". There is no need to clear or set the "carry" before these operations; the "carry" is set for you after these operations, however, analogously to 8-bit ADC and SBC operations.

Seven operations employ indirect addressing: that is, they fetch or set a value in memory whose address is named by the contents of Rn, not the value of Rn itself. These ops also affect the contents of Rn, either incrementing or decrementing them, either before or after the operations, as described. The "@" symbol, denoting indirect addressing, is required for assembly by Merlin-Pro.

LD @Rn loads the "accumulator" with the 8-bit byte in the memory address named by Rn (the high 8 bits of the "accumulator" are just zeroed).

ST @Rn stores the low 8 bits of the "accumulator" into the memory address named by Rn.

LDD @Rn loads the "accumulator" with the 16-bit word residing in memory, in the usual lo-hi order, at the address named by Rn.

STD @Rn stores all 16 bits of the "accumulator" into memory, in the usual lo-hi order, starting at the address named by Rn. After each of these four operations, Rn is incremented - once after LD and ST, twice after LDD and STD, so that Rn now points to the next 8-bit or 16-bit piece of data in memory.

Note: be careful! These opcodes are confusingly named. The direct ops, LD Rn and ST Rn, deal with 16-bit words. The analogously named indirect ops, LD @Rn and ST @Rn, deal with bytes.

POP @Rn loads the "accumulator" with the single byte in the memory address named by Rn, but only after having decremented Rn once (the high 8 bits of the "accumulator" are just zeroed).

Similarly, **POPD @Rn** loads the "accumulator" with the 16-bit word residing, in the usual lo-hi order, starting at the memory address named by Rn, but only after having decremented Rn twice.

STP @Rn stores the low 8 bits of the "accumulator" into the memory location named by Rn, but only after having decremented Rn once.

The Sweet16 instruction set: branch ops

We come now to branch instructions, that is, instructions dealing with the path that the program is to follow. Exactly as with 6502 branching, Sweet16 branches are limited to a distance of backward 128 bytes or forward 127 bytes.

Three instructions perform an unconditional branch.

BR LABEL branches to the address named by LABEL. This is the closest thing Sweet16 has to a JMP instruction, but the limitation to jumping -128 to +127 bytes, but this is not usually a problem, because Sweet16 code is so compact.

BS LABEL branches to the address named by LABEL, but remembers the point from which the branch was made. A subsequently encountered command **RS** branches back to the instruction following the BS command. *(Editor: There's a joke in there somewhere...)* These are thus the equivalents of the 6502 JSR and RTS commands, used for calling and terminating subroutines, and, as with them, subroutines may be nested. Note, however, that it is up to the user to set R12 beforehand with the lowest (that's lowest!) address of a safe block of memory to be used to save the addresses from which BS commands are executed.

Eight instructions branch if certain conditions are met. These conditions have to do either with the value of the "carry" or with the value of the last register (including the "accumulator") which was directly referred to (called the "last result"). Thus, for the direct commands listed above, the value involved is that of the register Rn on or from which the operation was performed; for the indirect commands, and for ADD and SUB, the value involved is just that of the "accumulator".

However, such branches alone would provide no way to perform a comparison test between the "accumulator" and some other value. To take care of this, a command **CPR Rn** is implemented. This command actually subtracts the value in Rn from the value in the "accumulator", and it is the result of this subtraction (stored in R13) which is tested in a subsequent conditional branch command. The operation is thus analogous to the 6502 CMP instruction.

BZ LABEL branches to LABEL only if "last result" is 0.

BNZ LABEL branches to LABEL only if the "last result" is not 0. These are thus the equivalents, whether or not they follow a CPR command, of the 6502 BEQ and BNE commands.

BP LABEL branches to LABEL only if the "last result" is positive

BM LABEL branches to LABEL only if the "last result" is negative. A 16-bit word is considered positive if and only if the hi-bit of its hi-byte is not set. These commands can thus be used, whether or not they follow a CPR command, like the 6502 BPL and BMI commands.

BC LABEL branches to LABEL only if the "carry" is set

BNC LABEL branches to LABEL only if the "carry" is clear. These commands may be used after an ADD or SUB command); note also that after a CPR command, they are the equivalents of the 6502 BGE/BCS and BLT/BCC commands, respectively. (Note that you must use these commands immediately after the command which sets or clears the "carry", because all other ops clear the "carry".)

BM1 LABEL branches to LABEL only if the "last result" is -1 ($FFFF)

BNM1 LABEL branches to LABEL only if the "last result" is not -1.

Entering and leaving Sweet16 mode

During a 6502 program, to signify that the following code is Sweet16 code and is to be interpreted by the Sweet16 interpreter, execute a JSR SW16, where SW16 is the address of the Sweet16 interpreter. The interpreter will then read and execute all subsequent code, until it encounters the command RTN; at this point, control will be turned over to the 6502, starting with the byte after the RTN.

For debugging purposes, Sweet16 also recognises a BK command; this simply executes a 6502 BRK, sending you to the monitor. After examining or modifying memory, you may resume execution from the monitor at the instruction after the break by modifying R15 ($1E/1F) to the memory address at which the BK was encountered and calling for a GO from the address called INTERP in Listing 1 (e.g., type "319G"). (You have to know in advance what this address it; the monitor will not display it for you, but will display the address of Sweet16 BK subroutine instead.) To avoid having the BK occur on a subsequent pass, you may substitute for it a byte 0D, which is interpreted by Sweet16 as a NUL (= NOP).

The paradox resolved

As an illustration of these opcodes, and of the value of Sweet16, examine Listing 3. It performs exactly the same task as Listing 2!

See the improvements? First, Listing 3 occupies about half the code space of Listing 2. This is because all the register operations except for SET are only 1 byte long, thanks to Wozniak's ingenious coding method (see Table 1). Of course, we also have to occupy some memory with the Sweet16 interpreter; but clearly a program involving several Sweet16 routines would soon realise significant savings in space, and it should not be hard to find a place to stash the Sweet16 interpreter where we will not find its presence troublesome.

Second, and more important, Listing 3 was easy to write. In fact, I wrote it from start to finish, without errors, without ever having to go back to an earlier step and modify it!

So Sweet16 code is easy to write, easy to debug, and easy to read; it's compact, and it's incredibly powerful whenever you have to deal with 16-bit information.

Feel Sweet16 again

Listing 1 contains my version of the Sweet16 interpreter. It is based on Steve Wozniak's original published version, and for that reason alone is worth reading even if you don't intend to implement Sweet16 for yourself, because Wozniak's code is nothing short of brilliant. But my version also contains some minor improvements over Wozniak's original. It is better labelled and commented than his version was. It saves the contents of zero-page addresses to be used as Sweet16's "registers" on entry, and restores them on exit, so that you can use it in combination with Applesoft programs and BASIC.SYSTEM. It also includes a self-relocator, so that half the code can be hidden away in Page 3; of normal RAM, only Page 8 is ultimately occupied.

To use Sweet16, first type it in and assemble it, and save it as, say, SWEET16. When you want it in place in memory, `BRUN` it; this will cause the relocator to put part of the code into Page 3, the rest remaining in Page 8, and the program will then `RTS` to you. Now you can load and run assembly-language programs using Sweet16 code; any time your program does a `JSR SW16` (here, $300), the code that follows will be interpreted by Sweet16.

A number of modifications are possible. If you don't want to use Page 3, just omit the self-relocator and reassemble; in that case, the interpreter will reside in Pages 8 and 9, and calls to it will have to be made to $900. (You will then want to `BLOAD SWEET16` to put it into memory, not `BRUN` it.) If you don't want Pages 8 and 9 occupied, you can modify and reassemble the code to be located anywhere you like: the only important thing is that all of the code from the label `ROUTINZ` to the label `RTN` must be on a single page; otherwise the calculation of subroutine addresses, and the branching to those addresses, won't work.

You're not even confined to using $00-1F as your Sweet16 registers: any series of 32 consecutive 0-page locations will work (in fact, $F0-FF and $00-0F would work, since direct indexed zero-page addressing automatically wraps around from $FF to $00). Just redefine `R0`, and `STAT` and `PC` relatively to it. If you know you've got 32 consecutive 0-page locations absolutely free, you could also omit the saving and restoring operations, to obtain some extra microseconds; if you don't care about losing the contents of the 6502 registers across a Sweet16 call, you could omit the calls to the monitor SAVE and RESTORE routines as well.

Have fun with Sweet16: it just might be the fountain of youth that your assembly-language programs need!

Matt Neuburg, PhD
13150 Wenonah SE #121
Albuquerque, NM 87123
(505) 292-7811

Table 1 - Sweet16 Opcodes

All the branch opcodes, including `BS`, are of the form `0n`, where `n` determines which operation is to be performed, and must all be followed by a second byte giving the effective address to branch to, precisely as with 6502 code: the effective address is calculated as the distance, in bytes, from the byte following the byte containing the effective address. [Thus, `01 FD` would branch to the byte before the branch command; `01 FE` would loop indefinitely; `01 00` would continue as if nothing had happened; `01 01` would skip the byte following the branch command.] The operations `RS`, `RTN`, `BK`, and `NUL` are also of the form `0n`; they are single-byte codes. The register operations, except for `SET`, are also all single-byte codes, where the first (hi) hex digit signifies the operation, the second (lo) hex digit designates the register to be operated on (15 register operations, 16 registers). The `SET` operation requires 3 bytes, one for the operation and the register, two for the value to which the register is to be set, in lo-hi order.

1n	SET Rn	00	RTN
2n	LD Rn	01	BR
3n	ST Rn	02	BNC
4n	LD @Rn	03	BC
5n	ST @Rn	04	BP
6n	LDD @Rn	05	BM
7n	STD @Rn	06	BZ
8n	POP @Rn	07	BNZ
9n	STP @Rn	08	BM1
An	ADD Rn	09	BNM1
Bn	SUB Rn	0A	BK
Cn	POPD @Rn	0B	RS
Dn	CPR Rn	0C	BS
En	INR Rn	0D	NUL
Fn	DCR Rn	[0E and 0F default to NUL]	

Listing 1

```
*********************************************
*                                           *
*                  SWEET16                  *
*                                           *
*  based on the original by Steve Wozniak   *
* new dissassembly w/ clarified code & self-*
*  relocator, by Matt Neuburg, PhD   3/9/89 *
*                                           *
*********************************************

RO       EQU   $00 ;and $01, "accumulator"
STAT     EQU   $1D ;"status" regstr: contains
                   ;indx to place last result
                  ;but since this is Rn*2, bit 0
                  ; is free: so it holds "carry"
PC       EQU   $1E  ;and $1F, "program counter"
SAVE     EQU   $FF4A ;monitor rtn, save regs
RESTORE  EQU   $FF3F ;monitor rtn, restore regs
```

```

         ORG    $800

         JMP    RELOCATE;Note:OMIT if don't want
                      ;self-relocation into Page
3

*----------------
* subrtn for Sweet16 ops: LABELs are op names
*----------------

ROUTINZ  EQU    *   ;CODE FROM HERE TO RTN MUST BE
                    ;ALL ON A SINGLE PAGE

*----------------
* register ops: on entry, Y is 2*num of opcode,
*  X is index to Rn (i.e., it's 2*n)
*----------------

SET      JMP    SETZ      ;(no room here)

LD       LDA    R0,X      ;move Rn to R0
BK       EQU    *-1       ;i.e., 00=BRK (cute, eh?)
         STA    R0
         LDA    R0+1,X
         STA    R0+1
         RTS              ;STAT points to Rn already

ST       LDA    R0        ;move R0 to Rn
         STA    R0,X
         LDA    R0+1
         STA    R0+1,X
         RTS              ;STAT pts to Rn already

DCR      LDA    R0,X      ;decrement Rn
         BNE    :no
         DEC    R0+1,X
:no      DEC    R0,X
         RTS              ;STAT points to Rn al-
ready

ST@      LDA    R0        ;send lo byte of R0
putinc   STA    (R0,X)    ; to memory via Rn
         LDY    #0
zerostat STY    STAT  ;R0 holds last rslt, say so
INR      INC    R0,X   ;increment Rn
         BNE    :no
         INC    R0+1,X
:no      RTS        ;STAT pts to Rn already if INR

LD@      LDA    (R0,X);get 1 byte from mem via Rn
         STA    R0      ;give it to lo-byte of R0
         LDY    #0
         STY    R0+1 ;just one byte, zero hi-byte
         BEQ    zerostat;(always)set STAT, inc Rn

LDD@     JSR    LD@     ;do lobyte, inc Rn, STAT=0
         LDA    (R0,X) ;do hi-byte
         STA    R0+1
         JMP    INR    ;and do another inc Rn

STD@     JSR    ST@    ;do lobyte, inc Rn, STAT
         LDA    R0+1    ;do hi-byte
         STA    (R0,X)
         JMP    INR     ;and do another inc Rn

POP      LDY    #0      ;hi-byte 0 if simple po
         BEQ    poplo   ; and go do lo-byte
POPD     JSR    DCR     ;dec Rn
         LDA    (R0,X)  ;get hi-byte from mem
         TAY            ;save it in Y
poplo    JSR    DCR     ;dec Rn
         LDA    (R0,X)  ;get (lo-)byte from mem
         STA    R0      ;lo-byte into "acc"
         STY    R0+1    ;hi-byte into "acc"
stzandgo LDY    #0      ;R0 holds last result..
         STY    STAT       ;so say so
         RTS

STP@     JSR    DCR        ;decrement Rn
         LDA    R0        ;stick lo-byte of "ac
         STA    (R0,X)     ; in memory via Rn
         JMP    stzandgo   ;STAT := 0

SUB       LDY   #0        ;if SUB, result to R0
CPR       SEC            ;if CPR, Y=13*2 on ent
          LDA   R0        ;do lo-byte subtracti
          SBC   R0,X
          STA   R0,Y     ;put result in R0 or R
          LDA   R0+1      ;do hi-byte subtracti
          SBC   R0+1,X
finish    STA   R0+1,Y   ;put result in R0 or R
          TYA             ;place of result * 2
          ADC   #0        ;"carry" into bit 0
          STA   STAT      ;rec both in STAT reg
          RTS

ADD       LDA   R0       ;do lo-byte addition
          ADC   R0,X     ;DON'T CLC-may be seri
          STA   R0        ;result to R0
          LDA   R0+1      ;do hi-byte addition
          ADC   R0+1,X
          LDY   #0       ;result is to go into
          BEQ   finish    ;& let prev rtn finis

*----------------
* branch ops;on entry,Y=0,X is 2*num of opcod
*  acc holds num of register we are to examin
*----------------

BS        LDA   PC        ;(X=12*2 on entry)
          JSR   putinc    ; use R12 to pt to
stack
          LDA   PC+1      ; put ret addr into m
```

```
             JSR    putinc    ; incr R12 as we go
BR           CLC              ;guarantee branch
BNC          BCS    NUL       ;don't bra if carry set
adjstpc      LDA    (PC),Y    ;examine branch addr
             BPL    :no      ;if displacmnt pos, Y:=0
             DEY             ;if dsplacmnt neg, Y:=-1
:no          ADC    PC        ;clear carry guaranteed
             STA    PC        ;lobyte PC+displacement
             TYA              ;tricky 2's-complement
             ADC    PC+1      ;  addition:
             STA    PC+1  ;hibyte PC+(0 or -1)+carry
             RTS            ;whew! PC now correct

BC           BCS    BR   ;here's an easy one:
NUL          RTS        ;no carry, no bra (also NUL)

*----------------
* subroutine of remaining branch ops
*----------------

ldhi         ASL              ;Rn in acc: double it,
             TAX             ;use rslt as indx to Rn,
             LDA    R0+1,X    ;& fetch hi-byte of Rn
             RTS              ;& you also clrd carry

*----------------
* and here they are...
*----------------

BP           JSR    ldhi       ;examine hi half of Rn
             BPL    adjstpc     ;branch if positive
             RTS

BM           JSR    ldhi       ;examine hi half of Rn
             BMI    adjstpc     ;branch if negative
             RTS

BZ           JSR    ldhi       ;pick up hi half of Rn
             ORA    R0,X ;will give 0 iff both are 0
             BEQ    adjstpc     ;branch if so
             RTS

BNZ          JSR    ldhi       ;pick up hi half of Rn
             ORA    R0,X ;will give 0 iff both are 0
             BNE    adjstpc     ;branch if not
             RTS

BM1          JSR    ldhi       ;pick up hi half of Rn
             AND    R0,X ;will be FF iff both are FF
             EOR    #$FF ;is it? CMP might wrck crry
             BEQ    adjstpc     ;branch if so
             RTS

BNM1         JSR    ldhi       ;pick up hi half of Rn
             AND    R0,X ;will be FF iff both are FF
             EOR    #$FF ;is it? CMP might wrck crry
             BNE    adjstpc     ;branch if not
             RTS

RS           LDX    #24   ;12*2, pt to stack thru R
             JSR    DCR         ;dec stack ptr
             LDA    (R0,X)      ;pop hi-byte ret ad
             STA    PC+1
             JSR    DCR         ;dec stack ptr
             LDA    (R0,X)      ;pop lo-byte ret ad
             STA    PC
             RTS              ;that's all, PC is rea

RTN          JMP    RTNZ        ;(no room here)

             DS     \     ;fill out the page
                          ;so SW16 will be at start of
                          ; next page

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


             ORG    $300;OMIT if you don't want sel
                        ; relocation to Page 3

*----------------
* entry to Sweet16
*----------------

SW16         JSR    SAVE   ;preserve registers
             LDX    #31   ;preserve 32 0-page valu
:MOVEONE     LDA    R0,X
             STA    STORAGE,X
             DEX
             BPL    :MOVEONE

             PLA        ;initialize "program counte
             STA    PC  ; from calling address
             PLA
             STA    PC+1

INTLOOP      JSR    INTERP   ;interpret a command
             JMP    INTLOOP  ;again & again & agai

INTERP       LDA    #>ROUTINZ;get pg-num of subrtn
             PHA              ; & stuff on the stac
             INC    PC       ;incrmnt program count
             BNE    :NO      ;rdy to examine instructn
             INC    PC+1
:NO          LDY    #0       ;rdy for indrct addrssing
             LDA    (PC),Y  ;exmne opcde of instructn
             AND    #$0F     ;msk 1/2, leaving reg num
```

```
         TAX            ;and use it as X-index
         LSR            ;restore it
         EOR   (PC),Y ;considr only 1/2 opcode
         BEQ   TOBR ;if 0, branch instr: do it
                       ; (also BK, RTN, or NUL)

         STX   STAT ;sav regstr specifiction*2
                    ;so that if brnch follows,
                    ;we know what reslt to chk
         LSR          ;obtain opcode digit*2
         LSR
         LSR
         TAY          ; & use it as Y-index, to
         LDA  OPTBL-2,Y;get lobyt of subr addr
                  ; (first opcode is 1, 1*2=2)
         PHA          ;stick on the stack...
         RTS          ;and "jump" to subr
                      ; (the "Wozniak waltz")
                     ;with Y holding 2*num of op
                     ; & X indexing Rn

TOBR     INC   PC ;prepare to examn brnch addr
         BNE   :NO  ;(the subroutines will do
         INC   PC+1 ; actual examining)
:NO      LDA   BRTBL,X   ;get lobyte subr addr
         PHA              ;stick it on the stack
         LDA   STAT       ;examine "status reg"
         LSR              ;& prepare crry w/ it
         RTS      ;"jump" to subr w/ acc hlding
                         ; register-num where
                         ; "last result" is,
                      ; with Y=0, crry rdy, &
                         ;  X=2*opcode

*-----------------
* table of lo-bytes of addrs of opcode subrtn
*-----------------

OPTBL    DB    SET-1 ;2 tabls are interwoven
BRTBL    DB    RTN-1
         DB    LD-1
         DB    BR-1
         DB    ST-1
         DB    BNC-1
         DB    LD@-1
         DB    BC-1
         DB    ST@-1
         DB    BP-1
         DB    LDD@-1
         DB    BM-1
         DB    STD@-1
         DB    BZ-1
         DB    POP-1
         DB    BNZ-1
         DB    STP@-1
         DB    BM1-1
         DB    ADD-1
         DB    BNM1-1
         DB    SUB-1
         DB    BK-1
         DB    POPD-1
         DB    RS-1
         DB    CPR-1
         DB    BS-1
         DB    INR-1
         DB    NUL-1
         DB    DCR-1
         DB    NUL-1
         DB    NUL-1
         DB    NUL-1

*-----------------
* opcode subroutines that wouldn't fit into 1
*-----------------

* for RTN

RTNZ     PLA        ;pull ret addr from JSR INTE
         PLA        ; & throw it away
         LDA   PC ;put PC in temp storage
         STA   TEMPPC ; before we trash it
         LDA   PC+1       ; by restoring 0-pag
         STA   TEMPPC+1
         LDX   #31        ;restr 32 0-page val
:MOVEONE LDA   STORAGE,X
         STA   R0,X
         DEX
         BPL   :MOVEONE
         JSR   RESTORE  ;restore registers
         JMP   (TEMPPC) ;BYE! bk to 6502-land.
TEMPPC   DA    $0000      ;storage

* for SET: when we arrive, Y=2, X indexes Rn

SETZ     LDA   (PC),Y ;get hibyte of const (y=
         STA   R0+1,X ;put in hi-byte of reg
         DEY              ;(y=1)
         LDA   (PC),Y     ;get lobyte of const
         STA   R0,X       ;put in lobyte of re
         TYA              ;set acc=1
         SEC             ;cheatsies:need to add
         ADC   PC        ;add 2 to program ctr
         STA   PC        ;(a 3-byte instruction
         BCC   :DONE
         INC   PC+1
:DONE    RTS             ;PC rdy for nxt instr

STORAGE  DS    32       ;for saving 32 0-pg val

RELEND   NOP

*-----------------
*-----------------
* relocate second part of file to Page 3
```

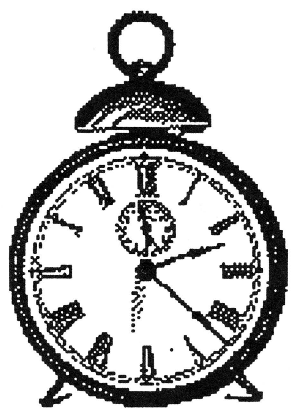
Listing 2

```
*******************************
*                             *
* EXAMPLE OF A TASK INVOLVING *
* 16-BIT OPERATIONS           *
*                             *
*******************************

* WARNING: this is NOT a complete program!
*          Do NOT try to enter and run it!

FILEPK    EQU   $2000 ;strt of packed file in mem
FILEUNPK  EQU   $4000 ;where unpked file will go
PKPTR     EQU   $FC         ;and $FD, pointer #1
UNPKPTR   EQU   $FE         ;and $FF, pointer #2

          ORG   $8000

*---------------
* First, we've got to put into 0-page memory
*  the file addresses, for indirect addressing
*---------------

UNPACK    LDA   #<FILEPK   ;lo-byte...
          STA   PKPTR
          LDA   #>FILEPK   ;...hi-byte
          STA   PKPTR+1
          LDA   #<FILEUNPK ;lo-byte...
          STA   UNPKPTR
          LDA   #>FILEUNPK ;...hi-byte
          STA   UNPKPTR+1

*---------------
* Set Y=0, for indirect addressing
*---------------

          LDY   #0

*---------------
* Run through FILEPK, looking for DLE=$FF
*---------------

ADVANCE   LDA   (PKPTR),Y  ;look at byte of FILEPK
          CMP   #$FF       ;is it DLE?
          BEQ   FEEDBLNK   ;=> yes, go handle
          STA   (UNPKPTR),Y ;no, feed to FILEUNPK
          JSR   INCPK      ;inc PKPTR
          JSR   COMPARE    ;see if we're at EOFPK
          JSR   INCUNPK   ;we haven't, inc UNPKPTR
          BNE   ADVANCE    ;loop back, always

*---------------
* Found a DLE, send the right number of blanks
*---------------

FEEDBLNK JSR    INCPK     ;inc PKPTR, & fetch..
         LDA    (PKPTR),Y  ;# of blanks to fee
         TAX              ;use it as index
         LDA    #$20       ;(SPACE)
ONEBLNK  STA    (UNPKPTR),Y;send blnk to FILEUN
         JSR    INCUNPK    ;inc UNPKPTR
         DEX               ;enough blanks sent
         BNE    ONEBLNK    ;No, go send anothe
         JSR    COMPARE    ;done w/ blanks, EC
         JSR    INCPK      ;no, so inc PKPTR..
         BNE    ADVANCE    ;...& loop back, always

*---------------
* Subroutine for incrementing PKPTR
*---------------

INCPK    INC    PKPTR      ;lo-byte...
         BNE    :NOTHI
         INC    PKPTR+1    ;...hi-byte
:NOTHI   RTS

*---------------
* Subroutine for incrementing UNPKPTR
*---------------

INCUNPK  INC    UNPKPTR    ;lo-byte...
         BNE    :NOTHI
         INC    UNPKPTR+1  ;...hi-byte
:NOTHI   RTS

*---------------
* Subroutine for comparing PKPTR with EOFPK
*---------------

COMPARE  LDA    PKPTR+1    ;hi-byte...
         CMP    EOFPK+1
         BNE    :NOTLO
         LDA    PKPTR   ;...lo-byte
         CMP    EOFPK
         BEQ    DONE  ;they're eql, go finish
:NOTLO   RTS          ;they're not equal, retu

*---------------
* Finish up, LENUNPK := UNPKTPTR-FILEUNPK+1
*---------------

DONE      JSR   INCUNPK    ;here's the +1
          SEC           ;and now we'll subtrac
          LDA   UNPKPTR    ;lo-byte...
          SBC   #<FILEUNPK
          STA   LENUNPK
          LDA   UNPKPTR+1  ;...hi-byte
          SBC   #>FILEUNPK
          STA   LENUNPK+1
```

```
         PLA  ;we got here from inside a subrtn
         PLA   ; so cancel return address
         RTS                ;The End
*----------------
* vars; in real life, these would have meaning
*----------------
EOFPK    DA    $2F37 ;whatever: last item addr
                     ; in packed file, + 1
LENUNPK  DA    $0000 ;result:len of unpked file
```

Listing 3

```
*********************************
*                               *
* SAME TASK INVOLVING 16-BIT    *
* OPERATIONS, USING SWEET16     *
*                               *
*********************************
* WARNING: this in NOT a complete program!
*           Do NOT try to enter and run it!
FILEPK   EQU   $2000 ;strt of pckd file in mem
FILEUNPK EQU   $4000 ;where unpacked file goes
SW16     EQU   $300  ;(or wherever Sweet16 is)
         SW          ;Sweet16 will be used below
         ORG   $8000
*----------------
* Call the Sweet16 interpreter
*----------------
UNPACK   JSR   SW16       ;start Sweet16 code
*----------------
* Initialise registers
*----------------
         SET   R1,FILEPK ;r1 will pt to pckd file
         SET  R2,FILEUNPK;r2 pts to unpacked file
         LD    R2
         ST    R7         ;save address in R7 too
         SET   R8,EOFPK   ;pt to variable EOFPK
         LDD   @R8   ;get value (R8 just temp)
         ST    R3    ;r3 will hold value EOFPK
         SET   R4,$00FF   ;r4 holds DLE
         SET   R5,$0020   ;r5 holds SPACE
*----------------
* Run through FILEPK, looking for DLE=$FF
*----------------
ADVANCE  LD    @R1      ;look at a byte of FILEI
         CPR   R4        ;is it DLE?
         BZ    FEEDBLNK   ;=> yes, go handle
         ST    @R2    ;no, just feed to FILEUNI
DONE?    LD    R1       ;consider ptr to FILEPK
         CPR   R3       ;is it EOFPK?
         BNZ   ADVANCE ;=> no, go do it again
         BR    DONE       ;=> yes, go finish u
*----------------
* Found a DLE, send the right number of blanks
*----------------
FEEDBLNK LD    @R1       ;get # of blanks to fe
         ST    R6          ;use it as index
         LD    R5          ;SPACE
ONEBLNK  ST    @R2         ;send it to FILEUNPK
         DCR   R6          ;decrement index
         BNZ   ONEBLNK;if not done, send anoth
         BR    DONE?    ;finished sending blank
                        ;go see if we've rchd E
*----------------
*Finish,LENUNPK:=R2(ptr 2 unpk'd fil)-FILEUNPK
*  (since R2 was incremented by last store)
*----------------
DONE     LD    R2
         SUB   R7       ;right answer now in R0
         SET   R8,LENUNPK ;(point to LENUNPK)
         STD   @R8       ;give answer to LENUNI
         RTN               ;leave Sweet16
         RTS               ;The End
*----------------
* vars; in real life, these would have meaning
*----------------
EOFPK    DA    $2F37      ;(or whatever)
LENUNPK  DA    $0000 ;result, length of unpack
                   ;file
```



IIgs Animation

Illusions of Motion, Part II

by Steven Lepisto

(Editor: This is the second in an extended semi-regular series on IIgs animation. Last time (March '90), Steve provided a "core" animation demo which he will modify with each new article.)

Last time, I presented a program that moved two images around the super hires screen. There was no background image, only blackness, and the objects interfered with each other when one passed over the other. But they were quick in their motions. This time, I am going to present the concepts of masks, background buffering, and shadowing. These techniques, when combined, will give us a more flexible, if somewhat slower, system of animation. That is always the trade-off: more flexibility generally means slower execution. However, when we can move an image across a multi-colored background without disturbing that background, perhaps the trade-off isn't quite so bad.

The two principle components of moving an image across a complex background are the concept of transparency and the concept of preservation of background. Taken together, it is possible to move an image across a complex background without disturbing either it or the image.

The Concept of Transparency

Transparency in an image is that part of the image that allows the background to show through (the way a window allows the outside to show through the wall). Transparency can be thought of as a "hole" in the image. So how can we put holes in images?

One way is to examine each pixel of the image as we plug it to the screen. If the pixel is to be transparent then don't put it on the screen, leaving the background alone. This method isn't all that fast but it's advantage is it saves the memory used by masks, which is the next method to talk about in making holes.

A mask is essentially a filter that allows certain areas of an image to be treated as transparent. It is combined with the image itself using the AND function. You then use the OR function to add the image to the background. ORing requires the transparent colors to be set to 0, otherwise the process will change the colors that are to shine through the transparent areas. The AND function gives us the ability to turn the transparent colors to 0.

The steps for putting the image into a background using a mask are:

1) Punch a hole in the background where the image will go. Doing this insures the bits in the image aren't influenced by the bits in the background.

2) Combine the image and mask to create a bit image with 0's in the places where the image is considered transparent. This way the bits in the background aren't influenced by the transparent bits in the image.

3) Add the combined image and mask data to the background using the OR function.

That's it. Here is a simple code fragment that shows the above steps.

```
lda     mask     ;step 1: punch hole in backgrounc
eor     #$ffff
and     background
sta     temp
lda     image    ;step 2: combine image and mask
and     mask
ora     temp     ;step 3: add image and mask to b
sta     background
```

To punch a hole in the background (that is, set to 0 all the areas in which the image will go), you need a "negative" of the mask: you are cutting a hole in the background where the image will be but you want to leave the background alone where the transparent areas in the image are. This is the exact opposite of how you want the mask to be applied to the image. So you create a negative of the mask by Exclusive ORing it with a pattern with all bits set to 1 (i.e., $ffff). You then AND the negative mask with the background and save the result off.

Combining the normal mask with the image will insure that all areas that are transparent are set to 0's.

To add the masked image to the background, take the masked image and OR it into the cut out background then store the result back in the background picture. All the 0 parts of the image will contain the background and all the 0 parts of the background will contain the image. Neat!

As a faster alternative, if the image already has 0's where you want the transparency to be, pre-inverse (or make negative before using) the mask then use the following code fragment:

```
lda     background
and     mask
ora     image
sta     background
```

This is much shorter and definitely quicker because there are fewer steps being taken. The only drawback to this approach is that color 0 (which is represented by a nybble with all 0's in it) can never be used in the images except as a transparent color. Sometimes this can be a serious drawback, but usually it isn't much of a problem.

To get around the problem of losing the use of color 0 as a color, the following technique can be used (the mask is not made negative but is normal for the image):

```
lda     image
eor     background
and     mask
eor     background
sta     background
```

Not much longer than the previous method and though it's a little slower, it is more flexible. The new PLOT_IMAGE routine uses this technique. The advantage of this method is when you make the mask for the image, you can say that any color of the mask is transparent. So if you create an image using color 6 as the transparent color (for whatever reason) you can create a mask that makes color 6 transparent and the above code will work quite nicely.

And that is how to use a mask to create a transparent color in an image.

The Concept of Preserving Background

Now, you may have been wondering how it is possible to erase the image once it has been added to the background using masks and ORing. That's a very good question. There are a couple of techniques that can be used. One is to keep a virgin copy of the background picture somewhere safe and whenever you need to erase something from the background being used, you copy from the virgin background those areas needing restoring. The advantage of this is it is faster than the next technique I'm going to describe. The disadvantage is it uses more memory—a lot more memory.

Another technique, and the one in use in the example program, is to preserve only enough background where the image will go before plotting the image. Then all you have to do is copy that buffered piece of background back from where you got it and the image will be erased. This way you only need to save and restore a rectangle the same size as the image being plotted which can save memory (over buffering the entire background), especially if you only have a few objects to deal with. However, it does add a step to the whole process of animation which can slow things down.

A third technique which is generally not very useful for most animation chores these days is the use of Exclusive OR. This technique provides a way of plotting an image with transparent areas of color 0 and removing that image from the background without needing to buffer the background. Exclusive OR is a function that will toggle bits on and off. If you do the same EOR process twice in a row with the same data, the image will be erased and the background restored to normal. This is very fast technique with only one serious disadvantage: the colors in the image and background tend to combine in really weird ways. However, EOR can be used for interesting effects or in areas where the background is mono-colored and color conflict won't arise.

How To Use Transparency and Buffered Background Without Flicker

Okay, to bring the whole process of transparency and buffering together, here is a simple algorithm for animation:

1) buffer (save) background where image will go
2) draw image into the background using a mask
3) erase the image by copying the buffered background on top of image
4) move the image
5) repeat steps 1 to 4.

There is only one problem with the above: it takes time to buffer and move an object and you will notice that there are moving and buffering steps between the erase and draw procedures. This means that the erase procedure will cause flicker because the time taken for the move and buffer procedures causes the eye to see the erasing process. How to cope with this?

We need a way of erasing and redrawing that isn't visible to the user... specifically, a way of drawing and erasing on an invisible screen while another is visible then with the flip of a switch show the new screen while drawing and erasing on the first screen, which is now invisible. Back and forth, back and forth, always working on the invisible screen. Because the eye would never see the erasing of the image, there would be no flicker. This technique is called page-flipping and we can't do it in super hires on the IIgs.

However, there is a way to sort of page-flip. This technique takes advantage of a hardware feature of the IIgs called *shadowing*. By using shadowing, a byte put in one area of memory will automatically be copied or shadowed to another area of memory. The super hires screen at address $e12000 is where the video circuitry gets the information to display the screen on the video monitor. The memory at $012000 is shadowed onto the memory at $e12000 but isn't directly connected to the video circuitry. When you write a byte to the memory starting at $012000 with shadowing turned on, it will automatically be copied to a corresponding point in the memory at $e12000. Since it is possible to turn off this shadowing function, you can write to the memory at $012000 and not have it show up at $e12000. This gives us the invisible screen to draw and erase on.

Now, just by turning on shadowing doesn't cause the shires screen at $012000 to magically appear at $e12000. Only when data is written to $012000 while the shadowing is on will it appear at $e12000. So a simple solution is to just read from the memory at $012000 and immediately write it back to the same point. Do this with the shadowing turned on and the image is made visible.

This is a lot slower than true page-flipping where you would simply hit a switch and the hardware would instantly start showing the second page. However, shadowing does provide a means of doing complex, flicker-free animation at a reasonable speed. One nice thing about shadowing is we take advantage of the higher memory speed in the shadowed memory (the memory at $e12000 always runs at 1MHz while the shadowed memory runs at over twice that speed).

So the steps for flicker-free animation with shadowing are:

1) turn shadowing off
2) buffer background from the shadowed screen where image will go
3) draw image into the background on the shadowed screen using a mask
4) turn shadowing on
5) copy the portion of the shadowed screen where the image is to itself
6) turn shadowing off
7) erase the image by copying the buffered background back to the shadowed screen
8) move the image
9) repeat steps 2 to 8.

True, this is more complex than the previous method, but it opens up the door to bunches and bunches of complex animation.

Updating the Experimental Program

Here is a list of changes to the program printed last time which will add support for masks and shadowing.

1) Make the changes indicated in listing one. The lines to change are marked; the additional lines are there to position the changes correctly.

2) Enter the new routines given in listing two.

3) Enter the mask and buffer data in listing three.

4) Finally, replace the old PLOT_IMAGE routine with the one in listing four.

Don't forget to create a new macro file for the finished source code.

In Conclusion

Animation is a complex process on a computer, even in its simplest form. However, with perseverance and experimentation, you too can create Illusions of Motion.

Things To Experiment With

1) In Plot_Setup, comment out the line "moveword #$01;shires_adrs+2" to see the action without the effects of shadowing. That is flicker. (Note, the program supplied on disk has different instructions for showing flicker. See the comments in the source code at the beginning of the program.)

2) Rewrite the heart of the Plot_Image code to use the plotting idea of:

```
lda    background
and    mask
ora    image
sta    background
```

Note that the masks will have to be inversed by hand (change all 0's to F's and all F's to 0's).

3) The Show_Image routine can be optimized a bit by eliminating the addition to screen_ptr at the end of the loop. See if you can work it out.

4) For the ambitious types, this program still won't work properly with velocities greater than 2. See if you can figure out where the problem lies (hint, the problem is in only one routine).

There are always different ways to apply the techniques I've described. The program code I wrote for this series isn't optimized for high speed animation. In fact, there are techniques I haven't described that are even faster than those given here. However, I have striven for ease of understanding over speed of execution *(Editor: for now)*. The code I gave here is quite good for many animation projects, however. Feel free to play with it, change it, use it as is. I am by no means the only person with knowledge of computer animation — I'm always learning something new about it!

Listing one:

In the following source code fragments, add the lines marked with a + at the end. Some of the routines have been truncated. This is indicated by ".....". *(Editor: Steve has reprinted some portions of the original animation engine here for continuity. You only need to add those lines marked with a '+').*

```
       dum   $00
deref_ptr ds   4
rowadrs_table ds 4
screen_ptr ds  4
image_ptr ds   4
mask_ptr ds    4                    +
buffer_ptr ds  4                    +
        dend


image_height ds MAXIMAGES*2
image_width ds MAXIMAGES*2
image_bytewidth ds MAXIMAGES*2
image_adrs ds   MAXIMAGES*4
mask_adrs ds    MAXIMAGES*4         +
buffer_adrs ds MAXIMAGES*4          +


Animate  jsr    init_images
         jsr    init_boundaries
         jsr    shadowon            +
         jsr    make_an_image       +
         jsr    shadowoff           +
:1       jsr    draw_images
         jsr    show_images         +
         jsr    erase_images        +
         jsr    move_images
         lda    #1
         jsr    pause_a_moment
         jsr    read_key
         bcc    :1
         rts


draw_images stz image_index

         .....

         lda    image_adrs,y
         sta    image_ptr
```

```
        lda    image_adrs+2,y
        sta    image_ptr+2
        lda    mask_adrs,y           +
        sta    mask_ptr              +
        lda    mask_adrs+2,y         +
        sta    mask_ptr+2            +
        lda    buffer_adrs,y         +
        sta    buffer_ptr            +
        lda    buffer_adrs+2,y       +
        sta    buffer_ptr+2          +
        jsr    buffer_image          +
        jsr    plot_image
        inc    image_index
        lda    image_index
        cmp    number_of_images
        bcc    :1
        rts


init_images ldx #0

        .....

        lda    def_image,y
        sta    image_adrs,y
        lda    def_image+2,y
        sta    image_adrs+2,y
        lda    def_mask,y            +
        sta    mask_adrs,y           +
        lda    def_mask+2,y          +
        sta    mask_adrs+2,y         +
        lda    def_buffer,y          +
        sta    buffer_adrs,y         +
        lda    def_buffer+2,y        +
        sta    buffer_adrs+2,y       +
        inx
        inx
        cpx    #MAXIMAGES*2
        bcc    :1
        txa
        lsr
        sta    number_of_images
        rts


def_height da  15,15
def_image adrl basic_image_1,basic_image_2
def_mask adrl  basic_mask_1,basic_mask_2     +
def_buffer adrl buffer1,buffer2              +


plot_setup ~GetPortLoc #shireslocinfo
        ~GetAddress #1
        pulllong rowadrs_table
        moveword #$01;shires_adrs+2          +
        rts
```

```
dostartup .....

        ~QDStartUp tool_dpage;#$00;#0;ProgramID
        bcs    :x
        lda    tool_dpage
        clc
        adc    #$300
        sta    tool_dpage
                                                  +
* Allocate shadow screen memory for our use.      +
                                                  +
      ~NewHandle #32768;PrivateID;#$c013;#$012000
        pla                                       +
        pla                                       +
        bcs    :x
        jsr    plot_setup
```

Listing two:

Add these entire routines to the code.

```
* Disable shadowing of the shadow screen.

shadow_register = $e0c035

ShadowOff ldal shadow_register
        ora    #%1000
        stal   shadow_register
        rts

* Enable shadowing of the shadow screen.

ShadowOn ldal  shadow_register
        and    #%1111_1111_1111_0111
        stal   shadow_register
        rts

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

* Draws a multi-color background picture on the
shadow
* screen on which to move the images.

make_an_image ~GetPortLoc #savelocinfo
        ~SetPortLoc #shireslocinfo
        stz    mai_loop_index
:1      lda    mai_loop_index
        asl
        tax
        phx
        ~SetSolidPenPat rect_color,x
        pla
        asl
        asl
```

```
        clc
        adc   #<rectangles
        tax
        lda   #^rectangles
        adc   #0
        pha
        phx
        _PaintRect
        inc   mai_loop_index
        lda   mai_loop_index
        cmp   #6
        bcc   :1
        ~SetPortLoc #savelocinfo
        rts

mai_loop_index ds 2

rect_color da  2,6,6,8,12,7
rectangles da  0,0,200,320
        da    10,10,100,100
        da    10,220,100,310
        da    130,20,180,300
        da    80,140,120,180
        da    178,260,190,270

* Used to save original port locinfo

savelocinfo ds 16


* Copy the regions in which the images were drawn
* themselves with shadowing turned on. This causes
* images to become visible on the shires screen.

show_images stz image_index
:1      lda   image_index
        asl
        tax
        asl
        tay
        lda   image_bytewidth,x
        sta   plot_bytewidth
        lda   image_height,x
        sta   plot_height
        lda   xposition,x
        sta   plot_xpos
        lda   yposition,x
        sta   plot_ypos
        jsr   show_image
        inc   image_index
        lda   image_index
        cmp   number_of_images
        bcc   :1
        rts
```

```
* Erase images on the shadow screen by copying th
* contents of the buffers which hold the bkground
* under the image.  This loop must be reverse of
* drawing loop else overlapping images wouldn't b
* properly erased.

erase_images lda number_of_images
        beq   :x          ;nothing to show
        dec
        sta   image_index
:1      lda   image_index
        asl
        tax
        asl
        tay
        lda   image_bytewidth,x
        sta   plot_bytewidth
        lda   image_height,x
        sta   plot_height
        lda   xposition,x
        sta   plot_xpos
        lda   yposition,x
        sta   plot_ypos
        lda   buffer_adrs,y
        sta   buffer_ptr
        lda   buffer_adrs+2,y
        sta   buffer_ptr+2
        jsr   erase_image
        dec   image_index
        bpl   :1
:x      rts


* Buffer background under image in preparation of
* drawing image. Buffered background is used to
* erase the image later on.

buffer_image lda plot_ypos
        asl               ;Y -> index
        tay
        lda   plot_xpos
        lsr               ;pixels to bytes
        clc
        adc   [rowadrs_table],y
        sta   screen_ptr
        lda   shires_adrs+2
        sta   screen_ptr+2
        ldx   plot_height
:row_loop ldy  #0
:byte_loop lda [screen_ptr],y
        sta   [buffer_ptr],y
        iny
        iny
        cpy   plot_bytewidth
        bcc   :byte_loop
        lda   buffer_ptr
        clc
        adc   plot_bytewidth
```

```
         sta    buffer_ptr
         bcc    :1
         inc    buffer_ptr+2
:1       lda    screen_ptr
         clc
         adc    shires_byte_width
         sta    screen_ptr
         dex
         bne    :row_loop
         rts


* Copy a rectangle of shadow screen to itself with
* shadowing turned on.  This causes the region
where
* image was drawn to become visible on the screen.

show_image jsr shadowon
         lda    plot_ypos
         asl               ;Y -> index
         tay
         lda    plot_xpos
         lsr               ;pixels to bytes
         clc
         adc    [rowadrs_table],y
         sta    screen_ptr
         lda    shires_adrs+2
         sta    screen_ptr+2
         ldx    plot_height
:row_loop ldy   #0
:byte_loop lda  [screen_ptr],y
         sta    [screen_ptr],y
         iny
         iny
         cpy    plot_bytewidth
         bcc    :byte_loop
         lda    screen_ptr
         clc
         adc    shires_byte_width
         sta    screen_ptr
         dex
         bne    :row_loop
         jsr    shadowoff
         rts

*----------------------------
* Erase an image from shadow screen by copying the
* contents of a buffer onto it.  The buffer holds
* background under the image.

erase_image lda plot_ypos
         asl               ;Y -> index
         tay
         lda    plot_xpos
         lsr               ;pixels to bytes
         clc
         adc    [rowadrs_table],y
         sta    screen_ptr
         lda    shires_adrs+2
         sta    screen_ptr+2
         ldx    plot_height
:row_loop ldy   #0
:byte_loop lda  [buffer_ptr],y
         sta    [screen_ptr],y
         iny
         iny
         cpy    plot_bytewidth
         bcc    :byte_loop
         lda    buffer_ptr
         clc
         adc    plot_bytewidth
         sta    buffer_ptr
         bcc    :1
         inc    buffer_ptr+2
:1       lda    screen_ptr
         clc
         adc    shires_byte_width
         sta    screen_ptr
         dex
         bne    :row_loop
         rts
```

Listing three:

Add these masks and buffers to the end of the program.

```
basic_mask_1 hex 0000000000000000  ;16 0's
         hex    0000000000000000
         hex    000ffffffffff000
         hex    00ffffffffffff00
         hex    00fffff00fffff00
         hex    00ffff0000ffff00
         hex    00fff000000fff00
         hex    00ff00000000ff00
         hex    00fff000000fff00
         hex    00ffff0000ffff00
         hex    00fffff00fffff00
         hex    00ffffffffffff00
         hex    000ffffffffff000
         hex    0000000000000000
         hex    0000000000000000

basic_mask_2 hex 0000000000000000
         hex    0000000000000000
         hex    0000000ff0000000
         hex    000000ffff000000
         hex    00000ffffff00000
         hex    0000ffffffff0000
```

```
        hex   000ffffffffff000
        hex   00ffffffffffff00
        hex   000ffffffffff000
        hex   0000ffffffff0000
        hex   00000ffffff00000
        hex   000000ffff000000
        hex   0000000ff0000000
        hex   0000000000000000
        hex   0000000000000000

* Enough buffer space for each image above
buffer1  ds    8*15
buffer2  ds    8*15
```

Listing four:

Replace the old plot_image routine with this one.

```
plot_image lda plot_ypos
        asl              ;Y -> index
        tay
        lda   plot_xpos
        lsr              ;pixels to bytes
        clc
        adc   [rowadrs_table],y
        sta   screen_ptr
        lda   shires_adrs+2
        sta   screen_ptr+2
        ldx   plot_height
:row_loop ldy  #0
:byte_loop lda [image_ptr],y
        eor   [screen_ptr],y
        and   [mask_ptr],y
        eor   [screen_ptr],y
        sta   [screen_ptr],y
        iny
        iny
        cpy   plot_bytewidth
        bcc   :byte_loop
        lda   image_ptr
        clc
        adc   plot_bytewidth
        sta   image_ptr
        bcc   :1
        inc   image_ptr+2

:1      lda   mask_ptr
        clc
        adc   plot_bytewidth
        sta   mask_ptr
        bcc   :2

        inc   mask_ptr+2
:2      lda   screen_ptr
        clc
        adc   shires_byte_width
        sta   screen_ptr
        dex
        bne   :row_loop
        rts
```

Basically Applesoft

Parms Away: Passing Parameters to Subroutines

by Robert Stong

In some high level computer languages, such as FORTRAN and Pascal, there is a technique known as passing parameters to a subroutine. This technique makes subroutines a much more powerful tool.

As an illustration, you can write a general subroutine which sorts an array X consisting of N elements. Then, in your program, with parameter passing, you can sort an array A with B elements by specifying that X is to be A and N is to be B. Without parameter passing available, one must either write the routine using the variables A and B or must move the contents of the array A to the array X so they can be sorted. The lack of passing ability makes the subroutine much less convenient.

Fortunately, Applesoft BASIC has a nice feature that will let you pass parameters to a subroutine. Many writers describe this feature as a flaw. The point is that

Applesoft BASIC will let you use long variable names, but only uses the first two letters of the name to identify the variable.

The way to use this feature is to write your subroutine using long variable names. When you are ready to use the routine with given parameters, it is only necessary to change the first two letters of each variable name. This can be accomplished quite simply by using a small machine language routine that recognizes long variable names and resets their first two letters. Effectively, the machine language routine is rewriting your subroutine using the variable names you have specified in your parameter list. Because the subroutine uses long names, variables can be identified by the last letters of their names, and the process can be repeated with other parameters.

To illustrate this method, I am including a demonstration program which has two subroutines. One subroutine creates a general menu from which the user selects options. The other subroutine forms the product of two matrices. These are quite typical general subroutines. The demonstration makes repeated use of these routines with different parameters.

Entering the program

If you have an assembler, enter the source code in Listing 1 and save the assembled object code as PARAMS. If you don't have an assembler, use the hex codes from Listing 2 and save the file with the command

```
BSAVE PARAMS, A$6000, L$E2
```

Enter the Applesoft program in Listing 3 and save it with the command

```
SAVE PAR.DEMO
```

Using Params with your programs

Obviously, if you want to use this machine language routine to do parameter passing in your own program, you need to know some details about it.

PARAMS is completely relocatable. I chose to assemble it at location $6000, but it can be loaded and run at any location. It does use memory at location $300 as workspace for manipulating variable names. If you are using page 3 for other purposes, you will want to change the location of TABLE (or change the hex 03's in locations $602A, $60A7, $60B2, and $60B8 to some other value, such as 61).

To use PARAMS, you write your Applesoft subroutine just as you always do except that any variable name you wish to use as a parameter must have at least three characters.
PARAMS ignores the first two characters and resets them based on your parameter list. All characters past the first two are used as identifying information.

The instruction to invoke PARAMS has the format

```
CALL address, line number, line number,
variable names.
```

The address is the load address of PARAMS (24576 if one uses $6000). The two line numbers are the first and last lines in which variable names will be changed. The list of variable names, separated by commas, are the new names desired within the line range. The order in which the names are listed is irrelevant, and you need only list a name if you want to change it.

Variable names for PARAMS must include the array and type identification. Thus ABXX, ABXX%, ABXX$, ABXX(, ABXX%(); and ABXX$( )are all treated as different variables: real, integer, string, and arrays of these. If ABXX occurs in the variable list, PARAMS will change all four-letter real variable names ending in XX to ABXX in the chosen line range.

PARAMS will not change anything occuring after REM, within quotation marks, or in a DATA statement.

PARAMS will change Applesoft function names. In the expression FN ABC(X), Applesoft uses exactly the same rules for identification as if ABC( )was an array of real variables. (Standard functions are tokenized so will not change, but user defined functions can and will change.) If you don't want to change a function name, you must be sure its name does not end in the same letters as are being used for any real array parameter.

WARNINGS

PARAMS accepts UV(), UV$, and UV% as valid variable names with three characters. If these were used in a parameter list, PARAMS will change all two-letter real arrays, integers, or strings to the given two-letter name. To avoid this, I would recommend always using names with at least four characters.

PARAMS does change the way in which your program is stored in memory. You should always be cautious when using such programs. Be sure to SAVE your program before you run it. (If you had a syntax error PARAMS might scramble your program.)

Using utilities which renumber an Applesoft program will probably not work with PARAMS. Most such utilities will not recognize the line numbers in the CALL statement. They would need to be changed manually.

Another system...

In volume 6, number 11 of *Nibble*, there was a feature article by H. Cem Kaner and John R. Vokey entitled "Subroutine Master". They provided an elaborate parameter passing system which was based on modification of the names stored in Applesoft variable space. The article included a discussion of many points to consider in such a system and the complications which arise.

While their system had some nice features that are not available with this method (named subroutines and local variables), I think you will find this system to be utterly simple. It avoids many of the complications.

Listing 1: PARAMS Poker

```
10  FOR X = 0 TO 226
20  READ ML
30  POKE 8192 + X,ML
40  PRINT  CHR$ (4)"BSAVE PARAMS.OBJ,A8192,L226"
99  END
10000  DATA  32, 190, 222, 32, 12, 218, 32, 26,
     214, 165
10010  DATA  155, 133, 6, 165, 156, 133, 7, 32,
     190, 222
10020  DATA  32, 12, 218, 230, 80, 144, 2, 230,
     81, 32
10030  DATA  26, 214, 32, 190, 222, 162, 0, 32,
     183, 0
10040  DATA  157, 0, 3, 32, 177, 0, 240, 7, 201,
     44
10050  DATA  240, 3, 232, 208, 241, 134, 25, 224,
     2, 176
10060  DATA  3, 76, 201, 222, 165, 6, 133, 8,
     165, 7
10070  DATA  133, 9, 160, 3, 200, 177, 8, 208, 26
     160
10080  DATA  0, 177, 8, 170, 200, 177, 8, 133, 9,
     134
10090  DATA  8, 197, 156, 208, 233, 228, 155, 208
     229, 32
10100  DATA  183, 0, 208, 184, 96, 201, 178, 240,
     226, 201
10110  DATA  131, 208, 11, 200, 177, 8, 240, 217,
     201, 58
10120  DATA  208, 247, 240, 206, 201, 34, 208, 11
     200, 177
10130  DATA  8, 240, 202, 201, 34, 208, 247, 240,
     191, 201
10140  DATA  65, 144, 187, 201, 91, 176, 183, 162
     0, 200
10150  DATA  177, 8, 201, 91, 144, 35, 228, 25,
     208, 171
10160  DATA  132, 26, 136, 177, 8, 221, 0, 3, 208
     17
10170  DATA  202, 224, 1, 208, 243, 136, 173, 1,
3,
     145
10180  DATA  8, 136, 173, 0, 3, 145, 8, 164, 26,
     208
10190  DATA  140, 201, 65, 144, 3, 232, 208, 207,
     201, 58
10200  DATA  176, 210, 201, 48, 176, 245, 201, 36
     240, 4
10210  DATA  201, 37, 208, 4, 232, 200, 177, 8,
     201, 40
10220  DATA   208, 190, 232, 200, 208, 186, 173
```

Listing 2: PARAMS Demo

```
100  PRINT  CHR$ (4);"BLOAD PARAMS"
110  DIM MA$(3),CO$(5),AA(5,5),BB(5,5),CC(5,5)
120  READ MT$: FOR I = 1 TO 3: READ MA$(I): NEXT
:MN = 3
130  READ CT$: FOR I = 1 TO 5: READ CO$(I): NEXT
:CN = 5
140  CALL 24576,500,555,MTTT$,MNRR,MAXX$(,MSSS:
GOSUB 500
150  ON MS GOTO 160,210,400
160  CALL 24576,500,555,CTTT$,CNRR,COXX$(,CSSS:
GOSUB 500
170  GR : HOME : COLOR= CS
180  FOR I = 0 TO 39: HLIN 0,39 AT I: NEXT
190  VTAB 22: PRINT "PRESS ANY KEY";: GET A$
200  GOTO 140
210  TEXT : HOME : HTAB 15: PRINT "MATRIX DEMO"
220  VTAB 3: HTAB 5: PRINT "This program will
compute the": PRINT "powers of the matrix:"
```

```
 230  PRINT : HTAB 16: PRINT "/";: HTAB 24: PRINT
"\"
 240  FOR I = 1 TO 5: HTAB 16: PRINT "|";: HTAB 24:
PRINT "|": NEXT
 250  HTAB 16: PRINT "\";: HTAB 24: PRINT "/"
 260  FOR I = 1 TO 3: FOR J = 1 TO 3:AA(I,J) = (J >
= I): VTAB 5 + 2 * I: HTAB 16 + 2 * J: PRINT
AA(I,J): NEXT : NEXT
 270  VTAB 14: PRINT "PRESS ANY KEY TO BEGIN ";:
GET A$:T3 = 3:P = 1
 280  HTAB 1: CALL  - 868
 290  VTAB 14: HTAB 10: PRINT "/";: HTAB 32: PRINT
"\"
 300  FOR I = 1 TO 5: HTAB 10: PRINT "|";: HTAB 32:
PRINT "|": NEXT
 310  HTAB 10: PRINT "\";: HTAB 32: PRINT "/":
PRINT
 320  VTAB 24: PRINT "PRESS ANY KEY TO STOP";: POKE
- 16368,0
 330  CALL
24576,600,625,BBXX(,AAYY(,AAZZ(,T3MM,T3NN,T3PP:
GOSUB 600
 340  FOR I = 1 TO 3: VTAB 13 + 2 * I: FOR J = 1 TO
3: HTAB 7 + 6 * J: PRINT BB(I,J);: NEXT : NEXT :P =
P + 1: VTAB 22: HTAB 19: PRINT P;"-th power": IF
PEEK ( - 16384) > 127 GOTO 380
 350  CALL 24576,600,625,CCXX(,BBYY(: GOSUB 600
 360  FOR I = 1 TO 3: VTAB 13 + 2 * I: FOR J = 1 TO
3: HTAB 7 + 6 * J: PRINT CC(I,J);: NEXT : NEXT :P =
P + 1: VTAB 22: HTAB 19: PRINT P;"-th power": IF
PEEK ( - 16384) > 127 GOTO 380
 370  CALL 24576,600,625,BBXX(,CCYY(: GOSUB 600:
GOTO 340
 380  POKE  - 16368,0: VTAB 24: HTAB 1: PRINT
"PRESS ANY KEY TO CONTINUE ";: GET A$
 390  GOTO 140
 400  TEXT : HOME : END
 410  DATA  "PARAMETER PASSING DEMO","COLORED
SCREEN","MATRICES","QUIT"
 420  DATA  "SCREEN
COLORS","MAGENTA","BLUE","VIOLET","GREEN","GRAY"
 491  REM  MENU ROUTINE
 492  REM  LINES 500-555
 493  REM  ENTER TTTT$=TITLE LINE
 494  REM        RRRR=# OF MENU ITEMS
 495  REM        XXXX$(=MENU ITEMS
 496  REM        SSSS=SELECTION MADE
 497  REM  LOCAL: H,I,X
 500  TEXT : HOME : HTAB 21 -  INT ( LEN (MTTT$) /
2): PRINT MTTT$
 505 H = 3: FOR I = 1 TO MNRR: IF H < 3 +  LEN
(MAXX$(I)) THEN H = 3 +  LEN (MAXX$(I))
 510  NEXT :H = 21 -  INT (H / 2)
 515  FOR I = 1 TO MNRR: VTAB 2 * I + 11 - MNRR:
HTAB H: PRINT I;") ";MAXX$(I): NEXT :I = 1
 520  VTAB 22: PRINT "ARROWS OR NUMBER TO CHANGE"
PRINT "<RETURN> TO SELECT"
 525  VTAB 2 * I + 11 - MNRR: HTAB H: INVERSE :
PRINT I;") ";MAXX$(I);: NORMAL : POKE  - 16368,0
 530 X =  PEEK ( - 16384): IF X < 128 GOTO 530
 535  POKE  - 16368,0:X = X - 128
 540  HTAB H: PRINT I;") ";MAXX$(I): IF X = 13 TH
MSSS = I: RETURN
 545  IF 48 < X AND X < 49 + MNRR THEN I = X - 48
 550 I = I + (X = 10) - (X = 11): IF I < 1 OR I >
MNRR THEN I = (I > MNRR) + MNRR * (I < 1)
 555  GOTO 525
 591  REM  MATRIX MULTIPLICATION
 592  REM  LINES 600-625
 593  REM  ENTER XXXX(=PRODUCT
 594  REM        MMMMxPPPP=ITS SIZE
 595  REM        YYYY(=FIRST FACTOR
 596  REM        NNNN=ITS COLUMNS
 597  REM        ZZZZ(=SECOND FACTOR
 598  REM  LOCAL: I,J,K
 600  FOR I = 1 TO T3MM: FOR J = 1 TO T3PP
 605 CCXX(I,J) = 0
 610  FOR K = 1 TO T3NN
 615 CCXX(I,J) = CCXX(I,J) + BBYY(I,K) * AAZZ(K,J
 620  NEXT : NEXT : NEXT
 625  RETURN
```

The Merlin Maniac

Making a List & Checking It Twice

by Steve Stephenson

So, you already know how to use the List Manager. Fine. I'd like to show you a few ways to extend your list power. In this article, I present two low-level 'hook' type routines that are designed to show you how to customize your lists. I've also thrown in a method to add a professional touch to your programs. This bag of tricks contains:

* A custom draw routine
* A custom compare routine
* A routine to detect double clicking

Note: These routines work equally well, whether you are using regular controls or the new extended controls.

Drawing the Member

The List Manager will draw your list members for you, provided they contain nothing but text in either pString or cString format. For many lists, this is acceptable; however, if you want something a little fancier, you will need to provide a custom draw routine.

The procedure I use in CustomDraw consists of:

1) my standard opening;
2) an attempt to skip drawing if the member is clipped;
3) erasing and redrawing the member;
4) adding selection marking;
5) and my standard closing.

The attempt to skip drawing a member (step 2) is adapted from GS Tech Note #74. It is designed primarily for scrolling, when the List Manager tries to redraw all visible members. In reality, all members may not need to be redrawn. The rect of the member to be drawn is passed to the draw routine, and if no part of that rect lies within the clip rect, you may skip drawing that member. This speed up trick is not needed on a small list like this sample, but, a list that has a complex draw routine or a lot of members visible would benefit from it.

The heart of this drawing routine is the code that actually puts something on the screen. The subroutine, DrawMember, shows one way of putting out something other than plain text. For this example, I use a combination of an icon, a text string, and a column of numbers.

The trickiest part of putting an icon in the list is getting the bits in the displayMode word correct. Since the easiest way to show a selected (highlighted) member is to invert it, you must take both the normal and inverted image into consideration when you decide the mode for _DrawIcon. With black and white icons, I have had the best success with a displayMode of $0001 and the icon mask identical to the image. My example uses color and because of the way my mask and mode are setup, the inverted icon reverses all colors. My advice is to experiment with different combinations of masks and modes.

By the way, if you want to use larger icons, it's a simple matter of using a larger value for listMemHeight. While you're at it, there's no reason to prevent you from drawing multiple lines of text in the member, or using a different font, or anything else that feels right.

The only other thing worth mentioning in DrawMember is the method I use to get a column of numbers to line up. Since all numbers take the same width, it would seem that nothing special is needed to align them; however, the space character is narrower than numbers and if you use the Integer Math tool set, you will inevitably get numbers with leading spaces. The easiest way I've found to right-justify the column of digits is to make all characters temporarily the same width with _SetFontFlags. Be sure to return to normal (proportional) afterward, or all following members will be drawn in this style. This brings up an important point: the List Manager does not save and restore any GrafPort stuff for you, so be careful what you change when you do your drawing.

The only thing needed to complete the custom drawing is to display the member with the proper highlighting. While this could be done during the actual drawing phase, I chose to apply it afterward. Information about whether the member should be shown inverted, dimmed or normal is contained in the upper three bits of the memFlag byte (see the comments in the listing under the label DrawSelect).

Sorting a List in Any Order

The _SortList call will arrange your list in ascending alphabetical order. As the List Manager knows nothing about what's in your data structure, it simply starts with the first byte and progresses through the data until it either reaches a difference or the end of the data. In fact, your only choice when using this feature is whether to use pStrings or cStrings (indicated in bit 0 of listType).

But with a custom compare routine, you can arrange your list in forward or reverse order using any part of a member's data. Don't be concerned with the sort algorithm that the List Manager uses; all you need to provide is a comparison routine that tells the List Manager which of two members you want to appear first. The List Manager passes you the pointers to the two members to compare and expects your decision to be returned in the carry flag. The List Manager then arranges the pointers in the array so they are in the order you want; when it comes time to draw the list, the draw routine is passed each pointer in turn from this array.

In the routine CustomCompare, I show a method of sorting on either of two fields in the data structure. After the standard opening, I convert (or dereference) the addresses that the List Manager passes into actual addresses of the member's data. Since I don't need the pointer that was passed after conversion, I recycle the direct page space by dereferencing it 'in place'.

The CompareSize routine simply indexes into the data structure and compares the integers. If they are different, the carry is set for the List Manager. If they are the same, I set it up to have the name field be the tie breaker.

The CompareName routine is fairly straightforward string manipulation. The only non-obvious part is the length checking at the top of the loop; this is done to ensure that when all of a short string matches the beginning of a longer string, the shorter string comes first.

Reversing the sort order is trivial: just flip-flop the state of the carry bit.

The Double Click Short Cut

By convention, double clicking may be used as a short cut. When used with a list control, the short cut might typically be linked to a default button. For example, in the tool call _SFGetFile, double clicking one of the file names in the list would produce the same effect as single clicking followed by a click in the Open button. While double- (and triple-) click information is now returned in the extended TaskMaster record, introduced with System Disk 5.0, my code is useful if your application isn't able to use TaskMaster.

For a double click to exist, two consecutive clicks must happen within the specified time limit and be close enough in location to be considered in the same item.

Detecting a double click begins with a mouse down event in the window. After tracking the click and making sure it's a hit in the list, the routine DoubleClick takes over. As the user may change the double click interval at any time (via the control panel), I start off by getting the latest setting. I divide the X coordinate by the height of a list line to get the number of the line where the click happened. I calculate the interval time between this and the previous click by using the system tick count from the Event record. Now, armed with the time and location information, I can determine whether this is a good double click. The check against ListCount traps clicks in the empty part of the list when there are fewer members than the maximum that can be displayed.

After calling the short cut routine (which I leave up to you), I clear the previous click time (forcing an impossibly large interval) to prevent the very next click from successfully falling through the routine again.

```
*==============================================
* Some list support routines                  *
*                                             *
* Copyright 1990 by Ariel Publishing          *
* and Steve Stephenson. Some rights reserved. *
*==============================================
         use   1/tool.equates/e16.event
         use   1/tool.equates/e16.window

WindowPtr ext             ;owning window
```

```
Event    ext               ;event record

*=============================================
listView =      5          ;max # viewable
listMemHeight = 10         ;height of one
lTop     =      10         ;where to draw list

ListRecord
:top     dw     lTop
:left    dw     10
:btm     dw     listView*listMemHeight+2+lTop
:rt      dw     300
ListCount
         dw     3          ;total # in list
         dw     listView   ;# viewable
         dw     %10        ;pString, single
         dw     1          ;start @ #1
         adrl   0          ;list ctl handle
         adrl   CustomDraw ;draw routine
         dw     listMemHeight ;member height
         dw     5          ;ptr array size
         adrl   ListPointers ;ptr array
         adrl   777        ;refcon
         adrl   0          ;color

*=============================================
* Uses speedup routine from GS Tech Note #74
CustomDraw
         phb               ;save B
         phk               ;reset B
         plb
         phd               ;save D
         tsc               ;reset D
         tcd
* what the dpage-in-stack looks like:
         dum    1          ;stk ptr
:d       ds     2          ;saved D
:b       ds     1          ;saved B
:rtl     ds     3          ;caller's rtn addr
clip:rect                  ;ptr to clip Rect
theEntry                   ;addr or item data
clipHandle                 ;Clip Rgn handle
listHandle adrl 0          ;ctl handle
memberPtr adrl 0           ;ptr to item
rectPtr  adrl   0          ;ptr to item's Rect
         dend

         ~GetClipHandle    ;to clip region
         PullLong clipHandle

         ldy    #2         ;deref handle
         lda    [clipHandle],y
         tax
         lda    [clipHandle]
         sta    clip:rect
         stx    clip:rect+2

* is this member's top below clip bottom?
         lda    [rectPtr]
         dec
         ldy    #region+bottom
         cmp    [clip:rect],y
         bcs    NoDraw     ; yes, not visible

* is this member's bottom above clip top?
         ldy    #bottom
         lda    [rectPtr],y
         inc
         ldy    #region+top
         cmp    [clip:rect],y
         bcc    NoDraw     ;  yes, not visible

         ~EraseRect rectPtr ;visible, clr old

         lda    [memberPtr] ;deref the data
         sta    theEntry
         ldy    #2
         lda    [memberPtr],y
         sta    theEntry+2

         ldy    #left      ;get Rect lt & btm
         lda    [rectPtr],y
         sta    memberRectLeft
         iny
         iny
         lda    [rectPtr],y ;btm
         sta    memberRectBtm

         jsr    DrawMember

DrawSelect
* now fix it's selected appearance (as req)
         ldy    #4
         lda    [memberPtr],y ;selected byte
         and    #%1110_0000 ;only valid bits:
* xxx0_0000 List Mgr's selection bits
* |||________1=inactive (dimmed, can't select)
* ||_________1=disabled (dimmed, selectable)
* |__________1=selected (inverted, if enabled)

* 000 = active & enabled, but not selected
         beq    DrawDone   ;leave normal
* 100 = active & enabled, and selected?
         cmp    #%1000_0000
         bne    DrawDim    ; no, then dim it
         ~InvertRect rectPtr ;yes, highlight
         bra    DrawDone

* 110, 010, 001, 011 = dim
DrawDim
         ~SetPenMask #DimMask
         ~EraseRect rectPtr
         ~SetPenMask #NormMask
NoDraw
```

```
DrawDone
        pld                 ;restore D
* pull B & the RTL addr off temporarily
        plx                 ; B & rtl bnk
        ply                 ; rtl addr
* pop the stuff that was passed to us
        pla                 ; (listHandle)
        pla
        pla                 ; (memberPtr)
        pla
        pla                 ; (rectPtr)
        pla
* now put the B & RTL addr back onto stk
        phy                 ; rtl addr
        phx                 ; rtl bnk & B
* and exit to caller
        plb                 ;restore B
        rtl                 ;back to List Mgr


memberRectLeft dw 0
memberRectBtm dw 0
DimMask  hex   55,AA,55,AA,55,AA,55,AA
NormMask hex   FF,FF,FF,FF,FF,FF,FF,FF

*=============================================
* Completely (re)draw this member/item
DrawMember
        pea    #^Icons      ;icon addr bank
        lda    [theEntry]   ;first is icon #
        asl
        tax                 ;make index
        lda    Icons,x      ;lookup icon addr
        pha
        pea    %0000_1111_0000_0000 ;mode
        lda    memberRectLeft
        clc
        adc    #4           ;move in some
        pha
        lda    memberRectBtm
        sec                 ;center icon
        sbc    #8+1
        pha
        _DrawIcon

        lda    #34          ;tab over for text
        jsr    TabTo

        lda    theEntry+2
        pha
        lda    theEntry
        clc
        adc    #4           ;offset to string
        pha
        _DrawString

        ldy    #2           ;offset to size
        lda    [theEntry],y
        pha                 ;the integer
        pushlong #sizeStr+1 ;result str
        pushword #5         ;max 5 digits
        pushword #0         ;unsigned
        _Int2Dec            ;convert to ascii

        lda    #180         ;move to size col
        jsr    TabTo
        ~SetFontFlags #2 ;set tabular mode
        ~DrawString #sizeStr
        ~SetFontFlags #0 ;reset proportional

        rts

sizeStr  str    '00000k'

*=============================================
* Move pen to draw text. Pass offset in Acc.
TabTo
        clc                 ;total from left
        adc    memberRectLeft
        pha
        lda    memberRectBtm
        dec                 ;move up 2 for
        dec                 ; font base line
        pha
        _MoveTo
        rts

*=============================================
Icons
        da     Disk:ram
        da     Disk:hard
        da     Disk:90mm

*-----------------------
Disk:ram
        dw     $8000        ;color
        dw     8*12/2       ;size of icon
        dw     8            ;# lines down
        dw     12           ;# nibbles accross
        hex    ffff0000000f
        hex    ff00bbbbbb0f
        hex    f0b0b0b0b00f
        hex    f0bbbbbbbb0f
        hex    f0b0b0b0b00f
        hex    f0bbbbbbbb0f
        hex    f0000066660f
        hex    ffffff66660f
:mask
        hex    ffffffffffff
        hex    ffffffffffff
        hex    ffffffffffff
        hex    ffffffffffff
        hex    ffffffffffff
        hex    ffffffffffff
```

```
         hex    ffffffffffff
         hex    ffffffffffff
*-----------------------
Disk:hard
         dw     $8000        ;color
         dw     8*12/2
         dw     8
         dw     12
         hex    ffffffffffff
         hex    ff00000000ff
         hex    f0333333330f
         hex    f0333334430f
         hex    f0333333330f
         hex    f0333333330f
         hex    ff00000000ff
         hex    ffffffffffff
:mask
         hex    ffffffffffff
         hex    ffffffffffff
         hex    ffffffffffff
         hex    ffffffffffff
         hex    ffffffffffff
         hex    ffffffffffff
         hex    ffffffffffff
         hex    ffffffffffff
*-----------------------
Disk:90mm
         dw     $8000        ;color
         dw     8*12/2
         dw     8
         dw     12
         hex    fff0000000ff
         hex    ff0f0fff0f0f
         hex    ff0f0fff0f0f
         hex    ff0f00000f0f
         hex    ff0ffffff0f
         hex    ff0ff0000f0f
         hex    ff00f0330f0f
         hex    fff0000000ff
:mask
         hex    ffffffffffff
         hex    ffffffffffff
         hex    ffffffffffff
         hex    ffffffffffff
         hex    ffffffffffff
         hex    ffffffffffff
         hex    ffffffffffff
         hex    ffffffffffff

*=============================================
CustomCompare
         phb                 ;save B
         phk                 ;reset B
         plb
         phd                 ;save D
         tsc                 ;reset D
         tcd

* what the dpage-in-stack looks like:
         dum    1            ;stk ptr
:d       ds     2            ;saved D
:b       ds     1            ;saved B
:rtl     ds     3            ;caller's rtn addr
memberA  adrl   0            ;ptr to A
memberB  adrl   0            ;ptr to B
         dend

         ldy    #2           ;deref ptr
         lda    [memberA]    ; to addr
         tax
         lda    [memberA],y
         sta    memberA+2,s
         txa
         sta    memberA,s
         lda    [memberB]
         tax
         lda    [memberB],y
         sta    memberB+2,s
         txa
         sta    memberB,s

         lda    CompareMethod ;which?
         bne    CompareName

*-----------------------
CompareSize
         ldy    #2           ;compare sizes
         lda    [memberA],y
         cmp    [memberB],y
         bne    CompareDone ;if sizes same,
                             ;then use name
*-----------------------
CompareName
         shortax
         ldy    #4           ;get str lengths
         lda    [memberA],y
         sta    lenA
         lda    [memberB],y
         sta    lenB
:loop
         tya
         sec
         sbc    #4
         iny
         cmp    lenB         ;shorter first
         beq    :sec
         cmp    lenA
         beq    :clc
         lda    [memberA],y
         cmp    [memberB],y
         beq    :loop        ;next char
         bra    :done

:sec     sec                 ;B first
         bra    :done
```

```
:clc     clc              ;A first
:done    longax

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

lenA     dw    0
lenB     dw    0
CompareMethod dw 0

*=============================================
inContent ent
         pha              ;spc
         pushlong #theControl
         pushlong Event+owhere
         pushlong WindowPtr
         _FindControl
         pla              ;part hit
         bne   TrackIt
noHit                     ;none
         rts

TrackIt
         pha              ;spc
         pushlong Event+owhere
         pushlong #-1     ;use action procs
         pushlong theControl
         _TrackControl
         pla              ;in same part?
         beq   noHit      ; no, strayed

AfterTrack
         ~GetCtlRefCon theControl
         pla              ;item number
         ply              ;throw away

         cmp   #777       ;in the list?
         bne   noHit

*-----------------------
DoubleClick
         ~GetDblTime      ;get interval
         pulllong DblTime

         lda   Event+owhere+2
         sta   thePoint+2
         lda   Event+owhere
         sta   thePoint

         ~GlobalToLocal #thePoint
         ldx   #0
         lda   thePoint
         clc
         sbc   ListRecord+top
         bmi   :gotit
:divide  sbc   #listMemHeight
         bcc   :gotit
         inx
         bra   :divide
:gotit
         stx   currline

         lda   Event+owhen ;current 'when'
         tax
         sec
         sbc   prevtime   ;- previous 'when'
         sta   interval   ;= time between
         lda   Event+owhen+2
         tay
         sbc   prevtime+2
         sta   interval+2

         stx   prevtime   ;reset for
         sty   prevtime+2 ; next time

         ldx   currline   ;in the same line?
         cpx   prevline
         stx   prevline
         bne   Done       ; nope.

         cpx   ListCount  ;in range?
         bcs   Done       ; nope.

         lda   DblTime    ;quick enough?
         sec
         sbc   interval
         lda   DblTime+2
         sbc   interval+2
         bcc   Done       ; nope.

         jsr   DoItem
         stz   prevtime   ;prevent response
         stz   prevtime+2 ; to triple click
         bra   Done

DoItem
* ...insert your 'short cut' routine here
* (to be called on double-click)

Done
```

```
        rts

currline dw    0
prevline dw    0
thePoint dw    0,0
theControl adrl 0
DblTime  adrl  0
prevtime adrl  0
interval adrl  0
ThisWindow adrl 0

*=============================================
* Array of pointers somewhere in memory
ListPointers
        adrl  item1       ;ptr to item
        dfb   0           ;List Mgr's byte
        adrl  item2
        dfb   0
        adrl  item3
        dfb   0
*=============================================
* Individual item data somewhere in memory
ListMembers
item1
        dw    1           ;my icon #
        dw    $8000       ;size (in k)
        str   'Hard Disk'
item2
        dw    2
        dw    800
        str   '3.5" Disk'
item3
        dw    0
        dw    256
        str   'Ram Disk'
*=============================================
```

Soft Thoughts: *Why I Wrote Super-Patch In BASIC*

by John Link

John Link is the author of SuperPatch, the popular AppleWorks patch utility published by Q Labs. Section 4 of his manual for SuperPatch is entitled "Some Thoughts". This article, reprinted from that section of the manual with John's permission and slightly edited to fit this magazine's format, explains why he wrote SuperPatch 6.1 in plain old Applesoft BASIC rather than assembly language.]

At one time, in a fit of masochism, I had decided to rewrite SuperPatch in machine language. Machine language is not only macho, it is also more compact than BASIC. As I added more patches to SuperPatch, the compactness of machine language became more and more appealing. But really, what appealed to me most was the "advancedness" of machine language. You haven't arrived as a programmer until you write something in ML, just like you cannot be accepted in certain social circles until you say things in French. If I had stuck with my decision, SuperPatch would be much smaller than it is now. Obviously one reason for the hypothetical compactness would be the compactness of ML itself. Another reason would be that much of my time would have been devoted to writing code that could handle all the things that BASIC does for SuperPatch, instead of developing new patches. Fewer patches means less code, no matter what language is used.

What came from this excursion into "advancedness" is the realization that BASIC is as advanced as any other computer language. The "B" in BASIC could easily stand for "Best" instead of "Beginner's" in its full name: Beginner's/Best All-purpose Symbolic Instruction Code. BASIC is more "all-purpose" than any other language for the Apple II, and seems perfectly suited to a program like SuperPatch. What most SuperPatchers want is a reliable and easily understood method of applying their modifications to AppleWorks. BASIC is very reliable, and the time I might have spent "reinventing the wheel" went into making the interface easier to understand and the program more thorough in its ability to cope with all the possible software and hardware configurations used by AppleWorkers. My interface contains few "bells and whistles." You have to press letter keys instead of moving an inverse menu bar up and down the lists of patches. Inverse menu bars can be programmed in BASIC, but, in BASIC, they are too sluggish for my taste, and they eat up memory and code rapidly. But, thanks to easily accessed BASIC commands, SuperPatch is thorough in its examination of your AppleWorks disk, which seems far more important. While machine language executes much faster than BASIC, the slowest part of SuperPatch is the examination of your files to determinne the status of the patch areas. ML would not accelerate this process to any noticeable extent.

Using BASIC opens SuperPatch itself to examination. Those of you who are interested can easily follow the logic of the program by simply listing the code. You can also customize SP to suit yourself, if you want. Or, as some have done, you can develop patches to SuperPatch. Most of the key areas contain REM statements to guide those who have an interest in programming. This kind of openness is simply not possible with machine language, unless you do what Robert Lissner, the creator of AppleWorks, did, and publish extensive notes. Even then, BASIC, because of its nature as an interpreted language, remains more open than any other, and therefore more subject to the user's modification.

[SuperPatch 6.1 costs $39.95 and is available from Q Labs, 1066 Maryland, Detroit, MI 48230, (313) 331-0941.]

Just Like the Big Boys

AppleWorks-Style Line Input

by Tom Hoover

It seems like whenever I write an assembly language program, there's always a need for some type of "string input" routine. For a long time, I wrote a separate "custom" routine for each program, which was, obviously, an inefficient way to program. Meanwhile, I had fallen in love with Robert Lissner's AppleWorks input routine, which features intuitive command keys and a default input string. I finally got around to writing a similar routine of my own that could be linked into any program that I was writing. I present it to all of you in the hope it can save someone else some work.

The GetStr routine supports the same commands as the AppleWorks line editor. Although you can easily change them if you don't like them, I suggest you keep them for purposes of standardization. The commands are:

```
OA-E or Control-E   toggle between insrt/ovrstrke cursors
OA-Y or Control-Y   delete from cursor to end of line
Left Arrow          move the cursor one space to the left
Right Arrow         move the cursor 1 space to right
OA-Left Arrow       move to the beginning of the input line
OA-Right Arrow      move to the end of the input line
OA-Delete           delete the character under the cursor
Delete              delete the character preceding the cursor
Escape              restore the default string
Return              accept the entire input line
```

GetStr is designed for use with the 80-column screen and does not support wrap-around of the input line, limiting you to an eighty-character input line (or less, depending on screen placement). I haven't found either of these limitations a problem in my applications, but the routine could be enhanced to eliminate them.

To use the GetStr routine, your program needs to do the following:

1. Place the "default" input string at InBuffer ($200) in Pascal string format (with a leading length byte). If you wish to have a "blank" default string, store a "0" at InBuffer.
2. Set your desired horizontal and vertical cursor position (OurCH, CV).
3. Place the maximum length of the input string in the X register.
4. Place the desired prompt character into the Accumu-

lator.

5. `JSR GetStr`. The prompt character will be printed at the current cursor co-ordinates (OurCH, CV), followed immediately by the "default" string (if any), and allow the user to edit the string.

6. The routine will return the string at InBuffer, with a leading length byte.

For an example of how to call GetStr, see the GetStr.Demo program listing.

GetStr is well commented, so you shouldn't have any problem figuring it out, but here's a rundown on how it works.

Before entering the main loop, GetStr first displays the default input string on the screen. It then calls GetKey repeatedly until a Return keypress is detected. GetKey calls another routine called KeyHandler, which actually does most of the routine's work.

GetKey (along with FlashDelay) provides the flashing insert or overstrike cursor, and performs the task of getting a keystroke from the user. This keystroke is then passed to KeyHandler. You can change the flash rate of the cursor by changing the variable cFlash; larger numbers result in slower flash rates. You can also change the default cursor type by changing the variable 'cCursor'; set the high bit for insert, or clear it for overstrike.

KeyHandler handles each keystroke by calling the appropriate command routine or by storing it in the buffer and displaying it on the screen. KeyHandler uses two lookup tables, one containing the value of the key associated with a command, the other containing the address (less one) of the routine that handles that command. The address of the command is pushed onto the stack and a "funny jump" is performed with an RTS, an old 6502 trick that goes back to the original Apple II Monitor.

I've also included two linker files (one for Merlin 8 and one for Merlin 16 and 16+) that demonstrate how to link GetStr into your program, which you'll need to do to run the GetStr.Demo program. The Merlin 16 linker file handles the assembly and linking of both GetStr and GetStr.Demo. The Merlin 8 linker file must be loaded into Merlin, assembled, and saved to disk as LINK. You must then assemble GetStr and GetStr.Demo, then type LINK $2000 "LINK" from the Merlin 8 command mode. After saving the object file, you can change it to a SYS file. Notice that Merlin 16 makes the entire process much simpler.

I hope you find GetStr useful. With it, you can easily implement part of the standard AppleWorks user interface in your programs.

Listing 1

```
********************************************
*
* GetStr Input Rtn w/ "AppleWorks" Cmd Keys
*
* Copyright 1990 by Tom Hoover
* All rights reserved
*
* You may use this rtn in your programs,
* as long as appropriate credit is given.
*
* This rtn inspired by "command line" editor
* in ApplWks.I've never found an input rtn i
* any other program that I liked as much as
* the one in AWks; so, I wrote one
* w/ a similar cmd set that I could include
* in any program.  Thanks, Robert!
*
********************************************

          rel              ;relocatable file
          dsk   getstr.L
          xc    off  ;remove if not using
                     Merlin 16+

          lst   off
          tr    on
          tr    adr

CV        =     $25
BASL      =     $28
OurCH     =     $57b
InBuffer  =     $200
InBuffer2 =     $280
Key       =     $c000
Strobe    =     $c010
Page1     =     $c054
Page2     =     $c055
OAKey     =     $c061
TabV      =     $fb5b
COut      =     $fded



*==================================================
*
* To use...
```

```
*
*  1. x = ma len of the desired input string
*  2. a = desired prompt character
*  3.If you want a "default" input str, place
*    at InBuffer ($200) in Pscal str form (w/
*    a length byte).  If you want a "blank"
*    default string, store a "0" at InBuffer.
*  4. The prompt will print at the current
*     cursor (OurCH,CV), followed immediately
*     by the "default" string (if any).
* 5. You can use the following at the prompt:
*
* oa-e -toggl between insrt/ovrstrike cursors
* ctrl-e  -ditto
* oa-y    -delete from cursor to end of line
* ctrl-y  -ditto
* left    -move cursor one space to the left
* right   -move cursor one space to the right
* oa-left -move to the beg. of the input line
* oa-right-move to the end of the input line
* oa-DEL  -delete the char under the cursor
* DEL     -delete the char preceeding cursor
* ESC     -restore the "default" string
* RTN     -accept the entire input line
*
*  6. The sub will return with the string at
*     InBuffer, with a prefixed length byte.
*
*-----------------------------------



*==================================================
*
* GetStr - the "Main" rtn.  Call with
*   X = the max input len, A = the desired
*   cursor char.  The prompt will show at the
*   current cursor position (OurCH,CV).  This
*   assumes the 80 column screen and it does
*   not support "wrapping-around" the screen
*   edge, so placement of prompt is important
*   long str.A & X are trashed,Y is saved

GetStr   ent
         sty   yTemp   ;save Y in a temp var
         stx   MaxLength  ;max len of input

         jsr   PutChar ;Put prompt on screen
         inc   OurCH   ;inc cursor to nxt pos

         lda   OurCH
         sta   chOrig  ;save cursor coords

         ldx   InBuffer
:loop    lda   InBuffer,x ;move "default"
string to InBuffer2
         sta   InBuffer2,x ;easily 2 restor
         dex                  ;when user hits E
         bpl   :loop

         jsr  PrintBuffer;pr InBuffer to sc

:keyloop jsr   GetKey
         cmp   #$8d       ;is it a return?
         beq   :done      ;if yes, accept

         jsr   KeyHandler
         jmp   :keyloop

:done    ldy   yTemp      ;restore Y regist
         rts



*==================================================
*
* KeyHandler - interprets each keystroke, a
*   performs accordingly

KeyHandler
         pha              ;save char on stack

         cmp   #$7b   ;is it an "open-apple
         bge   :noConvert ;nope

         cmp   #$60      ;is it "lower-case
         blt   :noConvert ;nope

         and   #%11011111;convert to upper

:noConvert
         sta   Char       ;save char
         ldx   #$ff       ;initialize loop

:keyloop inx
         lda   :KeyTable,x
         beq   :noCmd     ;end of table

         cmp   Char       ;found it yet?
         bne   :keyloop   ;nope, so try aga

         pla ;it's a command, so clean up
             ;stack

         txa           ;transfer offset to A
         asl           ;double it
         tax           ;and transfer back to

         lda   :KeyAddrs+1,x ;do indirect jm
                by pushing
         pha        ;the address on the sta
         lda   :KeyAddrs,x
         pha
         rts          ;and then doing an
```

```
RTS


:KeyTable  ;our routine looks in this table...
         db    'E'         ;oa-E
         db    $9f&"e"     ;ctrl-E
         db    'Y'         ;oa-Y
         db    $9f&"y"     ;ctrl-Y
         db    $8          ;oa-left
         db    $15         ;oa-right
         db    $88         ;left
         db    $95         ;right
         db    $7f         ;oa-DEL
         db    $ff         ;DELete
         db    $9b         ;ESCape
         db    #0          ;end of table

:KeyAddrs  ;to find the address in this table
         da    :do_oaE-1  ;oa-E
         da    :do_oaE-1  ;ctrl-E
         da    :do_oaY-1  ;oa-Y
         da    :do_oaY-1  ;ctrl-Y
         da    :do_oaLeft-1 ;oa-left
         da    :do_oaRight-1 ;oa-right
         da    :do_Left-1 ;left
         da    :do_Right-1 ;right
         da    :do_oaDEL-1 ;oa-DELete
         da    :do_DEL-1  ;DELete
         da    :do_ESC-1  ;ESC


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

:noCmd
         pla           ;get original character
         cmp   #$a0
         blt   :99     ;less than a , so it's a
                       ;ctrl char, and we don't
                       ;want it

         pha

         lda   OurCH ;curr pos minus original
         sec         ;pos = current position
         sbc   chOrig;in string
         tax

         pla

         cpx   InBuffer   ;are we at the end?
         beq   :append ;yes, append next char


         bit   cCursor ;ck insrt/ovrstrk  mode
         bmi   :insert

:over
         inx
         sta   InBuffer,x ;put it in buffer

         jsr   PutChar    ;put it on screen
         inc   OurCH
         bne   :99        ;always

:append
         cpx   MaxLength
         bge   :99        ;too long

         inx
         stx   InBuffer
         sta   InBuffer,x ;put it in buffer

         jsr   PutChar    ;put it on screen
         inc   OurCH
         bne   :99        ;always

:insert
         inx
         stx   Temp       ;current position

         ldx   MaxLength
         cpx   InBuffer ;can we insrt anothe
         beq   :99      ;nope, at MaxLength

         pha

         lda   InBuffer
         tay
         sec
         sbc   Temp       ;current position
         tax
*  move everything over, so that we
:2       lda   InBuffer,y   ;can insert
         sta   InBuffer+1,y ;our character
         dey
         dex
         bpl   :2

         inc   InBuffer

         pla
         ldx   Temp
         sta   InBuffer,x ;put it in buffer

         jsr   PrintBuffer ;reprint input st
         inc   OurCH


:99      rts

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

:do_oaE
         lda   cCursor
```

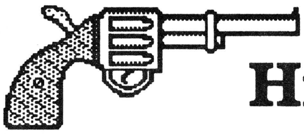
```
         eor    #%10000000 ;invert hi-bit
         sta    cCursor
         rts

*—————————————————

:do_oaY
         lda    OurCH
         sec
         sbc    chOrig
         sta    InBuffer ;truncate at curr pos
         jsr    PrintBuffer ;print entire str
         rts

*—————————————————

:do_oaRight
         lda    InBuffer ;goto end of input str
         clc
         adc    chOrig
         sta    OurCH
         rts

*—————————————————

:do_oaLeft
         lda    chOrig  ;beginning of input str
         sta    OurCH
         rts

*—————————————————

:do_Right
         lda    OurCH ;crsr to the right by one
         sec          ;character
         sbc    chOrig
         cmp    InBuffer
         beq    :done_Right
         inc    OurCH

:done_Right
         rts

*—————————————————

:do_Left
         lda    OurCH ;move cursor to left one
         cmp    chOrig     ;character
         beq    :done_Left
         dec    OurCH

:done_Left
         rts

*—————————————————

:do_oaDEL
         lda    OurCH
         sec
         sbc    chOrig
         cmp    InBuffer   ;end of the string
         beq    :done_oaDEL ;yes
         inc    OurCH
         bne    :do_DEL    ;always

:done_oaDEL
         rts

*—————————————————

:do_DEL
         lda    OurCH
         sec
         sbc    chOrig ;find curr pos in stri
         tax

         beq   :done_DEL ;no mo chars to dele

         cmp    InBuffer
         beq    :1

:delLoop lda    InBuffer+1,x ;move everything
               over to close
         sta    InBuffer,x ;up the space left
               by the deleted
         inx                   ;character
         cpx    InBuffer
         bne    :delLoop

:1       dec    InBuffer

         jsr    PrintBuffer
         dec    OurCH


:done_DEL rts

*—————————————————

:do_ESC
         ldx    InBuffer2  ;restore "default"
:ESCloop lda    InBuffer2,x
         sta    InBuffer,x
         dex
         bpl    :ESCloop

         lda    chOrig
         sta    OurCH

         jsr    PrintBuffer

         rts

```

```

*==============================================================
*
* PrintBuffer-prints contents of InBuffer, at
*   chOrig.  OurCH is preserved.

PrintBuffer
          lda   OurCH  ;save curr pos on stack
          pha
          lda   chOrig;set OurCH to 1st char of
          sta   OurCH ;the input str scrn pos

          ldx   InBuffer
          beq   :done;nothing to pr, then done
          ldx   #0

:loop     inx
          lda   InBuffer,x
          jsr   COut   ;print InBuffer to scrn

          cpx   InBuffer
          bne   :loop

:done     lda   #$1d       ;clear EOL
          jsr   COut

          pla
          sta   OurCH   ;restore curr crsr pos

          rts



*==============================================================
*
* PutChar - this routine "puts" char in "A"
* on the scrn at the current cursor position.

PutChar
          pha              ;save char on stack

          lda   OurCH  ;get horizontal pos
          lsr
          tay          ;put in main or aux mem?
          bcs   :mainpage

          sta   Page2      ;set to aux

:mainpage pla             ;get char from stack
          sta   (BASL),y   ;PUT the character
          sta   Page1      ;reset to main

          rts


*=========================================
*
* PickChar - "picks" the char from screen
* at curr cursor pos, and returns it in "A"

PickChar
          lda   OurCH  ;get horizontal pos
          lsr
          tay      ;pick from main or aux me
          bcs   :mainpage

          sta   Page2      ;set to aux

:mainpage
          lda   (BASL),y   ;PICK the charact
          sta   Page1      ;reset to main

          rts




*==============================================================
*
* GetKey-this gives flashing crsr, & return
* a keystroke in "A".  If char is "negative
*(bit 7 set), then OA key was NOT pressed.
* the char is "positive" (hi-bit clr), then
* the open-apple key was pressed.

GetKey
          jsr   PickChar  ;get char under cr
          sta   cChar
          and   #%01111111 ;clear hi-bit

          cmp   #$40
          blt   :storeit

          cmp   #$60
          bge   :storeit

        and  #%00111111;remap so prts invers
                        ;instead of MouseText

:storeit sta   cOver  ;store overstrike cur

:loop     bit   cCursor ;insert or overstrik
          bmi   :insert

:over     lda   cOver
          bne   :printit

:insert   lda   cInsert

:printit jsr   PutChar   ;put on the scree
          jsr   FlashDelay ;wait awhile
          bmi   :gotit

          lda   cChar
```

```
         jsr   PutChar
         jsr   FlashDelay
         bpl   :loop

:gotit   bit   OAKey
         bpl   :99
         and   #%01111111 ;OA, so clear hi-bit

:99      bit   Strobe

         pha              ;save it
         lda   cChar
         jsr   PutChar    ;restore org to scrn
         pla

         rts



*==============================================================
*
* FlashDelay - provides delay for the flashing
* cursor.

FlashDelay
         ldy   cFlash
         ldx   #0

:keyloop lda   Key
         bmi   :99

         dex
         bne   :keyloop

         dey
         bne   :keyloop

:99      rts


*===============================
*
* Various storage locations

MaxLength ds   1
Char     ds    1
Temp     ds    1
yTemp    ds    1
chOrig   ds    1

cFlash   db    #200;adjust the flash rate here
cInsert  db    "_"    ;insert cursor
cOver    db    ' ' ;overstrike cursor(inverse)
cCursor  db    $80 ;insert=$80 overstrike=0
cChar    ds    1  ;current char under crsr
```

```
*********************************
*                               *
* GetStr Input Routine DEMO     *
*                               *
* Copyright 1990 by Tom Hoover  *
* All rights reserved           *
*                               *
*********************************

         rel         ;relocatable file
         dsk   getstr.demo.L
         xc    off   ;remove if not Merlin 16

         lst   off
         tr    on
         tr    adr

CV       =     $25
BasL     =     $28
OurCH    =     $57b
InBuffer =     $200
InBuffer2 =    $280
Key      =     $c000
Strobe   =     $c010
Page1    =     $c054
Page2    =     $c055
OAKey    =     $c061
TabV     =     $fb5b
COut     =     $fded

GetStr   ext


*===============================
*
* Start

         lda   #1          ;turn on 80 column
         jsr   $c300

         lda   #10   ;put cursor where we wa
         jsr   TabV
         lda   #10
         sta   OurCH

         ldx   string ;default str to Inbuff
:loop    lda   string,x
         sta   InBuffer,x
         dex
         bpl   :loop

         ldx   #25     ;maximum input length
         lda   #">"       ;input prompt
         jsr   GetStr  ;call which allows th
             ;user to edit the "default" inp
             ;string, or to enter their own

```

```
         lda    #15   ;move cursor down 5 lines
         jsr    TabV
         lda    #10
         sta    OurCH

         ldy    #0
:MsgLoop lda    :Msg,y
         beq    :MsgDone

         jsr    COut        ;print a msg
         iny
         bne    :MsgLoop

:Msg     asc    "You typed: "00

:MsgDone
         ldy    InBuffer
         lda    #0
         sta    InBuffer+1,y;put 0" at end of o
                ;input string to use as a
                ;terminator in the following
                ;print routine (I never said
                ;this demo routine was elegant)

         tay
:MsgLoop2 lda   InBuffer+1,y
         beq    :Msg2Done

         jsr    COut        ;print "new" strin
         iny
         bne    :MsgLoop2

:Msg2Done

:keyLoop bit    Key         ;wait for a key
         bpl    :keyLoop
         bit    Strobe

         jsr    $bf00       ;quit
         db     $65
         da     quit_parms

         brk

quit_parms db   4
         ds     0


string   str    "This is a test"
```



Hired Guns

8/16 is providing a free service to all programmers: 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, Ariel Publishin, Box 398, Pateros, WA 98846 or send us E-Mail to R.W.LAMBERT. Note that we'll run your ad twice - approximately every other month for four months. You'll need to renew your request to continue.

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, desktop 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.

We'll run M- Z next month.

Warning! What follows really IS an advertisement. It just doesn't look like one (or pay like one - shucks).

An Advetorial by Ross W. Lambert, Publisher

You've heard of "Near Beer"? Well, this is nearly an ad. I couldn't face writing any more ad copy this month. Exclamation points make me tired after a while - all that excitement, if you know what I mean. Instead I decided I'd give you my thoughts on a few subjects related to a couple of our products (so this is still *sorta* an ad).

First, the compiled, 8 bit BASICs. We sell Micol Advanced Basic IIe/IIc and we'll order ZBasic for you if you want it (our price is $54.95 plus shipping from Zedcor). *But neither environment may be for you.*

Here's why: Just about every compiler I've seen has been frustratingly slow and unweildly unless you are developing on a system larger or faster than the target system. For example, if you are creating a program targeted at 64K machines, then the compiled languages are a pain to use unless you're using at least a 128K machine for development. Likewise, if you're putting together a 128K application, then you'd better have a RAM disk, a fast hard drive, and/or a IIgs. Furthermore, the compilers have so much work to do that we categorically *do **not** recommend them* for1.0 mhz machines.

The reason for this state of affairs is that language development systems typically have three separate parts; an editor, support routines or libraries, and the actual compiler which turns your text (source code) into machine code. If you are writing a 128K application using a 128K Apple, then the entire development system cannot fit into memory at the same time as your program. There is therefore a *ton* of disk access as segments get swapped in and out. This is true for both ZBasic and MAB IIe/IIc, and it can drive you nuts if you are used to working in Applesoft.

A better alternative for those of you with 128K (or less) machines without hard drives or RAM disks is the *Toolbox Series* from Roger Wagner Publishing. These pure assembly language extensions to Applesoft give you much of the advantages of the compiled BASICs without nearly as much hassle. You simply install the routine you want and then program normally in Applesoft (without the disk access horrors of compilation). We are not currently selling any RWP products (we're gradually moving *out* of software sales except for those products we develop ourselves), but you can certainly get more information or order directly from the good folks in El Cajon (619/442-0522).

Back at the ranch, if you have a RAM disk and a fast system, the compiled languages can really provide a high level programming environment with built-in text editors *and* generate fast machine code for maximal use of a smaller target machine. Thus you can probably coax a little more overall performance from a compiled BASIC program (or, in the case of ZBasic, get three our four times as much numerical accuracy in floating point computations). The compilers are really quite nice to program with on a Transwarped GS.

As for which of the compiled 8 bit BASICs is better, at this point I can only say that "It depends". For graphics related things (including text generation on the graphics screens), I like ZBasic better. For text based programs, I like MAB, the reason being that MAB allows you to use more of the 128K for variables. This means more data in memory. As for documentation quality, the nod definitely goes to ZBasic. As for quality of the text editors, MAB is light years ahead of ZBasic.

You may be wondering why we don't sell a "C" or a Pascal - as I mentioned earlier, we're backing away from non-Ariel software sales due to about 15 dozen conflicts of interest. Some of our stock is being liquidated by Kevin Thornton at KAT Systems (see their ad on page 14). We do hack in C around here (with varying degrees of success).

We *are* still selling MAB IIe/IIc and MAB GS, hence you can get in on a really great deal as part of our part of our liquidation sale. We wanna move these buggers now, so MAB IIe/IIc has been reduced to $59.95 and MAB GS has been slashed to $75.

We're still selling our homegrown products, of course, and we have several very exciting additional projects well under way. For now, though, **I think the best deal in the house is *8/16 on Disk***. We're talking serious value here, friends. Each and every month you get at least 500K of material - everything from the magazine plus a whole lot more. We have both an 8 bit and a 16 bit program selector/ file viewer/ graphics viewer to help you navigate (some folks are finding these gadgets useful in their own right). And our featured files so far have included the actual source code to: Floyd Zink's Binary Library Utility, Bruce Mah's File Attribute Zapper II, Parik Rao's Orca/APW developer's utilities, and more other goodies than you could imagine (how about multi-tasking on your IIe?). These disks are fun, educational, and *useful*. You are welcome to lift any routines or libraries you find and plop them right into your own projects (with the exception of a very few things where authors have indicated they wish otherwise). One year of the disk is $69.95, six months is $39.95, and three months is $21. Oh yeah, we'll sell you two years of the disk for $129.95. Any single disk is $8.00.

This has got to be one of the strangest ads on record, but if I had to typeset one more breathless 48 pt headline I wuz gonna puke.

Give us a call at 509/923-2249, or write to: Mike Rochip, c/o Ariel Publishing, Box 398, Pateros, WA 98846

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

The Sensational Lasers

Apple IIe/IIc Compatible

SALE $345 *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 programs! Take advantage of this exceptional value today........... ***$345***

Super High Speed Option!

only **$385**

The LASER 128EX has all the features of the LASER 128, plus a triple speed processor and memory expansion to 1MB $385.00

The LASER 128EX/2 has all the features of the LASER 128EX, plus MIDI, Clock and Daisy Chain Drive Controller $420.00

DISK DRIVES

- * 5.25 LASER/Apple 11c .... $ 99.00
- * 5.25 LASER/Apple 11e .... $ 99.00
- * 3.50 LASER/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 $620.00

THE SUPERSTAR a LASER 128 Computer with 14" RGB Color Monitor and the LASER 145E Printer $785.00

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 $129.00

USA 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

Your satisfaction is our guarantee!

Phone Orders: 1-800-654-5426
8 - 5 Mountain Time • No Surcharge on Visa or MasterCard Orders!
Customer Service 1-800-537-8596 • In Colorado (303) 938-9089
FAX Orders: 1-303-939-9839