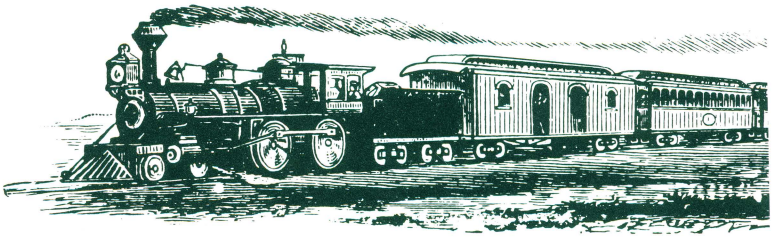
July, 1990
Volume 1, No. 5

8/16

The Journal of Apple II Programming $3.50

Kansas or Bust!



In this issue:

Computer	Title	Author	Page
both	The Publisher's Pen	Ross Lambert	3
	re: KCFestiveness, What's Cookin', Mind Control		
8 Bit	Speech Recognition	David Gauger	5
	re: a speech recognition hardware and software project		
IIgs	Illusions of Motion	Stephen Lepisto	27
	re: the third installment of Steve's IIgs animation series		
8 bit	MD-BASIC	Jay Jennings	32
	re: a review of the new Applesoft pre-processor		
IIgs	SouthPaw	Jason Blochowiak	35
	re: writing a permanent initialization file in APW and MPW IIgs		
both	Vaporware	Murphy Sewall	44
	re: the industry-wide ruminations of Murph the Magnificent		

Ariel Publishing
P.O.Box 398
Pateros, WA 98846
(509) 923-2249

Slide-On kits are $14.95 ea.
$12 ea. in quantities of 10+.

New kit restores your Apple IIGS

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. 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.

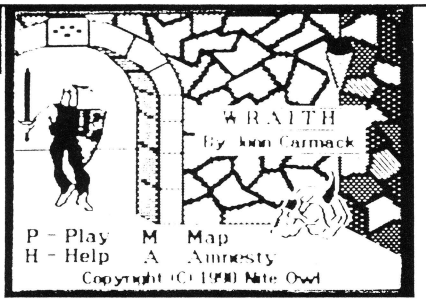
Typically, our customers have reported that the original equipment batteries have an average life expectancy of 2 to 3 years. This is about half as long as they were supposed to last. Slide-On replacement kits include Heavy Duty batteries which should provide for a longer battery service life.

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, and come with a full 90 day satisfaction guarantee.

New

WRAITH
Adventure Game

Special Introductory Price $9.95*



This graphic adventure game comes complete on a single 3.5 inch disk with on-screen instructions, a map, demo play option, and dungeons which were too vast and expansive to fit on 5.25" disks.

The object is to search out and destroy the evil WRAITH to save the mythical island of Arathia. To succeed at this quest the adventurer must fend off many monsters, learn magic spells, and buy weapons and armor to defeat the evil WRAITH.

Works on ANY Apple][with a 3.5" drive. It will have a retail price of $14.95. One of the best software values ever! * Offer expires 12/31/90

Font Collection *- The A2-Central staff has spent years searching out and compiling hundreds of IIGS fonts. These fonts are packed onto eight 3.5 inch disks. They work with IIGS paint, draw, and word processing programs. Includes a program to unpack them and an Appleworks data file.* $39

School Purchase Orders are Welcome.

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	$ 14.95	
	WRAITH Adventure	$ 9.95	
	Font Collection	$ 39.00	
		Kansas Sales Tax	
		Shipping & Handling	
		TOTAL	

Signature for Credit Card Orders

Please include $2 shipping and handling / $5 for overseas orders. Kansas residents add 6% sales tax.

Prices may Change without notice.

Please give us a call today at: **(913) 362-9898**
Photo-Copyable FAX: (913) 362-5798

Nite Owl Productions
5734 Lamar Avenue A
Mission, KS 66202
USA

(Cut & Paste Address Label)

8/16

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

Introductory subscription prices in US dollars:

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

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

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

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

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

Welcome to our first annual KansasFest issue! Yes, yes, I know, KansasFest is not the actual title of the A2-Central Developer's Conference, but it is such a truly festive event that it goes by that name more often than its proper one. What a joy to go hang out where the spirit of the Apple II lives on stronger than ever!

We're cooking

I get a lot of requests for Ariel Publishing to produce this kind of book or that kind of book or this kind of developer tool or that kind... I just wanted to let you know that we are working on several things that will be useful to y'all. We just can't rush 'em; as Don Lancaster said in *The Incredible Secret Money Machine*, "Iff'n it ain't cooked, don't serve it."

It is sage advice we're trying to follow.

Incidentally, when I determined to launch *8/16*, I was told that I'd need a quarter of a million dollars and a reserve that would let us run in the red for two years. By the time you read this, the magazine operations should have crossed over into the black.

This entire scenario was further highlighted in my mind when I had an interesting conversation with the president of a Macintosh software company. This fellow said, "I like you, Ross, and I want to see you succeed. Why are you messing around with the Apple II?"

I was my typical obsequious self, not wanting to offend. But I really should have said, "Let's compare balance sheets."

His company lost $200,000+ in the first quarter of 1990.

Propaganda Alert

Rajiv Mehta is one of the marketing geeks at Apple, Inc. I was reading a transcript of some statements he made about the Apple II recently . Although his remarks were generally pro-Apple II, I grew rather incensed about one little phrase. He prefaced one of his main ideas with the words, "Although there are limits to Apple II technology..."

I think Bill Mensch (developer of the 65C02, 65816, etc.) probably knows a little more about the hardware side of things than does marketing maven Rajiv. And "Wild Bill" (as I affectionately refer to him) has a vision for the Apple II that blows my mind. And if that is not enough, take a look at our hardware hacker column this month. What other computer has an architecture that permits you to develop hardware add-ons like this for *under $20?* I assure you that David's project and the software that goes with it have incredible commercial implications.

Mr. Mehta, unless you are prepared to explain in detail exactly *how* the Apple II is limited, then I suggest you keep your propaganda to yourself. I see your statement as a subtle attempt at mind control. It is apparent to me that some at Apple, Inc. *still* don't believe in the Apple II long term, and statements like yours are designed to get us to believe that we ought to move to a "better" computer after the next GS comes out, or perhaps before.

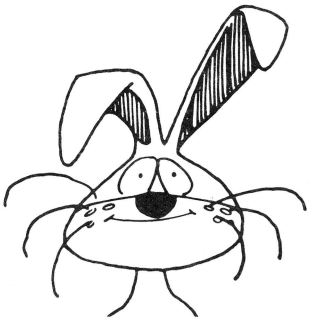
Poppycock.

If the rumors about the features of the next GS are even halfway true, then all the GS needs is competitive pricing and a real marketing commitment from the boys up at the big house.

I certainly recognize that, on the whole, Apple, Inc. is starting to rediscover the II. I appreciate John Sculley's comments about the "Macintosh IIgs". I appreciate the fine work done by the systems software group in the last year. I am anxiously awaiting the next GS CPU. I am not an ungrateful wretch. But don't kick my dog, slander my mother, or tell me the II series is technologically limited, Rajiv. It's future is only limited by your imagination.

The Weekend Hardware Hacker

Speech Recognition: Give Your Apple][-Ears!



by David Gauger

David is a music instructor at Oral Roberts University in Oklahoma. His obviously professional interest in sound combined with his love for the Apple II has synergetically produced this mind boggling project. I sincerely wish you could all have a chance to see this in action - it is quite startling to actually have your II respond to your voice.

Not only is it sheer fun, but it is also has commercial implications, especially for making programs accessible to the handicapped. I hope a couple of you take this ball and run with it.

Even if you don't, however, the software itself - especially the assembly module - has some very interesting techniques worth checking out.

== Ross ==

Communicating with a computer has never been easy from the human point of view. Since the earliest computers we've been forced to interact with them primarily through a typewriter-style keyboard. If your typing skills are anything like mine, you've often wanted to bypass the keyboard and speak directly to your Apple.

This column's project will enable you to do exactly that.][-Ears is a combination hardware/software system that makes it possible for you to add speech input to your own programs. You can literally speak to your computer and have it understand and act on the words you say.

The hardware is simple, easy to build, and interfaces to the game port. All the parts are available from your local Radio Shack for under $20.00. The system is organized as two machine language routines which are called from your own programs. These routines are loaded at $6000 (24576). Memory required for the routines and overhead typically runs about 3K of memory for 40 words. A program using the maximum 255 words increases the memory requirement to just under 10K. The system runs on any Apple][, but due to timing routines, accelerators will have to be turned off, and GS users will have to set the clock speed to "Normal".][-Ears is compatible with both ProDOS and DOS 3.3.

What's the Catch?

If this sounds too good to be true, it just might be, depending on what your expectations are.][-Ears will not allow you to dictate a letter into your Apple for editing and print out, for instance. It may even mis-identify words every so often. (All present day speech recognition systems make errors on a regular basis.) However,][-Ears is good enough to be quite useful in many areas. Applications for the physically handicapped, general experimentation, and hands-off operation of your computer are three obvious uses.

Though speech recognition is on the cutting edge of technology, perfect speech recognition is not a reality yet. It is being studied in several different areas of computer science, including the artificial intelli-

gence community, because so many factors are involved in understanding speech. Consider, for example, the fact that our speech tends to be connected. We go immediately from one word to the next with no space in between. Computers have a tough time deciding when one word ends and the next begins. After all, from a computer's point of view, words are just a succession of sounds.

Regional accents, background noise, speech impediments, and gender all contribute to the difficulties of computer speech recognition. Voice inflection is another problem the computer must deal with. As you speak, many factors change depending on the sentence. Are you saying a question, command, emphatic statement, or sleepy comment? Depending on the type of sentence, your voice inflection, loudness, speech rhythm and speed, syllable accent, etc. can all change drastically.

As an example, watch the way the word "left" changes in the following sentences:

- "The butter is to the left of the cheese in the refrigerator."
- "You mean we don't have any left??"
- (As barked out by a drill sergeant) "Left!— Left!— Left Right Left!"

"Computers have a tough time deciding when one word ends and the next begins".

All this is to point out that human language is highly variable. We very rarely say the same word in exactly the same way. Computers are exactly the opposite. They are rigid, unforgiving, and always do everything in exactly the same way. This basic incompatibility is part of what makes computer speech recognition such a difficult problem.

][-Ears Solutions

][-Ears seeks to minimize these problems by simplifying them. While the English language has many thousands of words,][-Ears has a practical limit of 15 or 20 and is most accurate with just a few. As for the connected speech problem,][-Ears accepts only one distinctly spoken word at a time.

To overcome regional accents, speech impediments, gender, etc.,][-Ears is "speaker dependent." This means the system must be pre-trained by the person whose words are to be identified. In other words,][-Ears will only identify words it has been trained to listen for. In addition, it will only understand the person who trained it. To train][-Ears, you say one word several times while][-Ears listens and records the data. If you like, this data can be stored on disk so retraining is not necessary every time you use the system.

To account for inflection and the variableness of speech,][-Ears incorporates a user adjustable "fudge factor" where each word is identified and passed back to the calling program with a score which indicates how sure][-Ears is that it correctly identified the word you said. The calling program can then decide whether the score is good enough to accept][-Ears suggested match as the correct identification of the word.

Break Out the Soldering Iron

To get your speech recognition system running, build the hardware according to the circuit diagram appropriate for your machine. The circuit is the same, only the connector to your particular flavor of Apple changes. Use Fig. 1 if you have an Apple][, or][+ (16 pin DIP connector). If you have a //c or //c+ with a DB-9 connector, use Fig. 2. If your machine is //e or //GS you can use either one.

The parts lists (Fig. 3 and 4) do not include a low cost dynamic microphone - the kind that come with cassette recorders, answering machines, etc. I'm assuming you've got one around the house already. Any low impedance (about 500 ohms) mic should work just fine. If you don't have one, Radio Shack has one of these also for about $10.00.

Schematic Diagram - 16 Pin DIP

Use with Apple][,][+, //gs

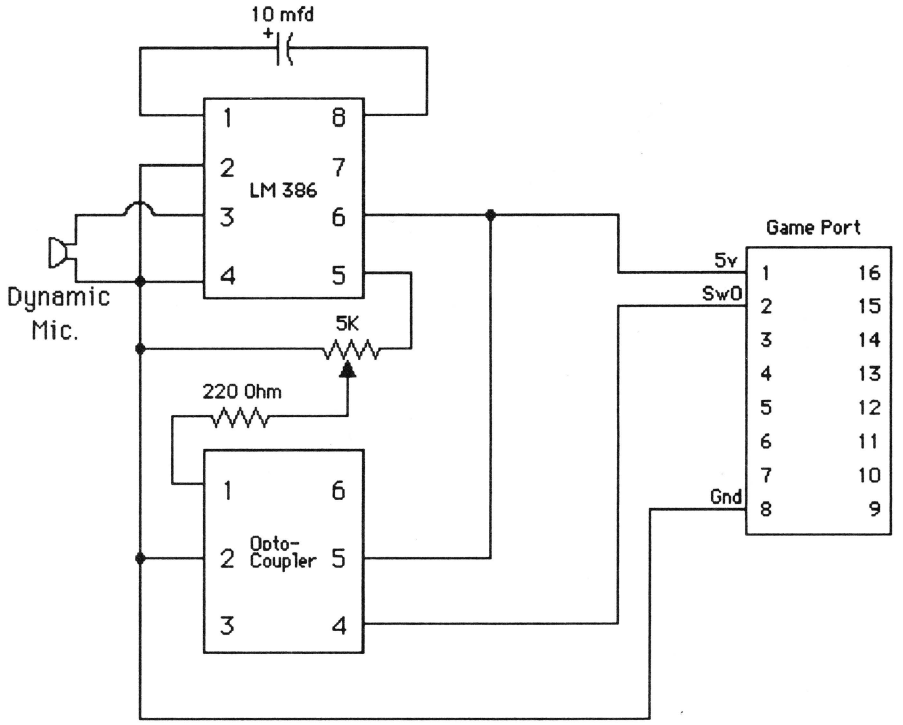


Figure #1

If you build circuits regularly, you'll already have many of the parts in your "junk box". Construction and component layout are not critical. Insulate all bare wires with electrician's tape and make sure you orient the chips correctly. The small indented dot on the top of the chip shows where pin one is. Pay particular attention when wiring up the connector to your Apple. We're dealing with low voltages here so shock is not a danger, but wrong connections or shorts could damage your Apple. Just inspect your connections carefully checking for stray bits of wire or solder that could cause problems.

To set the potentiometer (and to make sure the circuit is functioning correctly), plug in your microphone then type in and run this 2 line program:

```
10 IF PEEK(49249) >127 THEN PRINT "*";
20 GOTO 10
```

If the circuitry is functioning correctly, the program will come up in one of 2 states: either the screen will fill up with asterisks, or it will be doing nothing. The objective is to adjust the potentiometer to the point where the asterisks just stop filling the screen. If they're filling it, turn the "pot" (as it's called) until they just barely stop. If the asterisks weren't filling the screen, turn the pot the other way to get them to start, then back off just a hair. At this point, the asterisks should come on the screen only when you speak into the microphone.

In working with this hardware for a while, I've gotten the best results holding the microphone perpendicu-

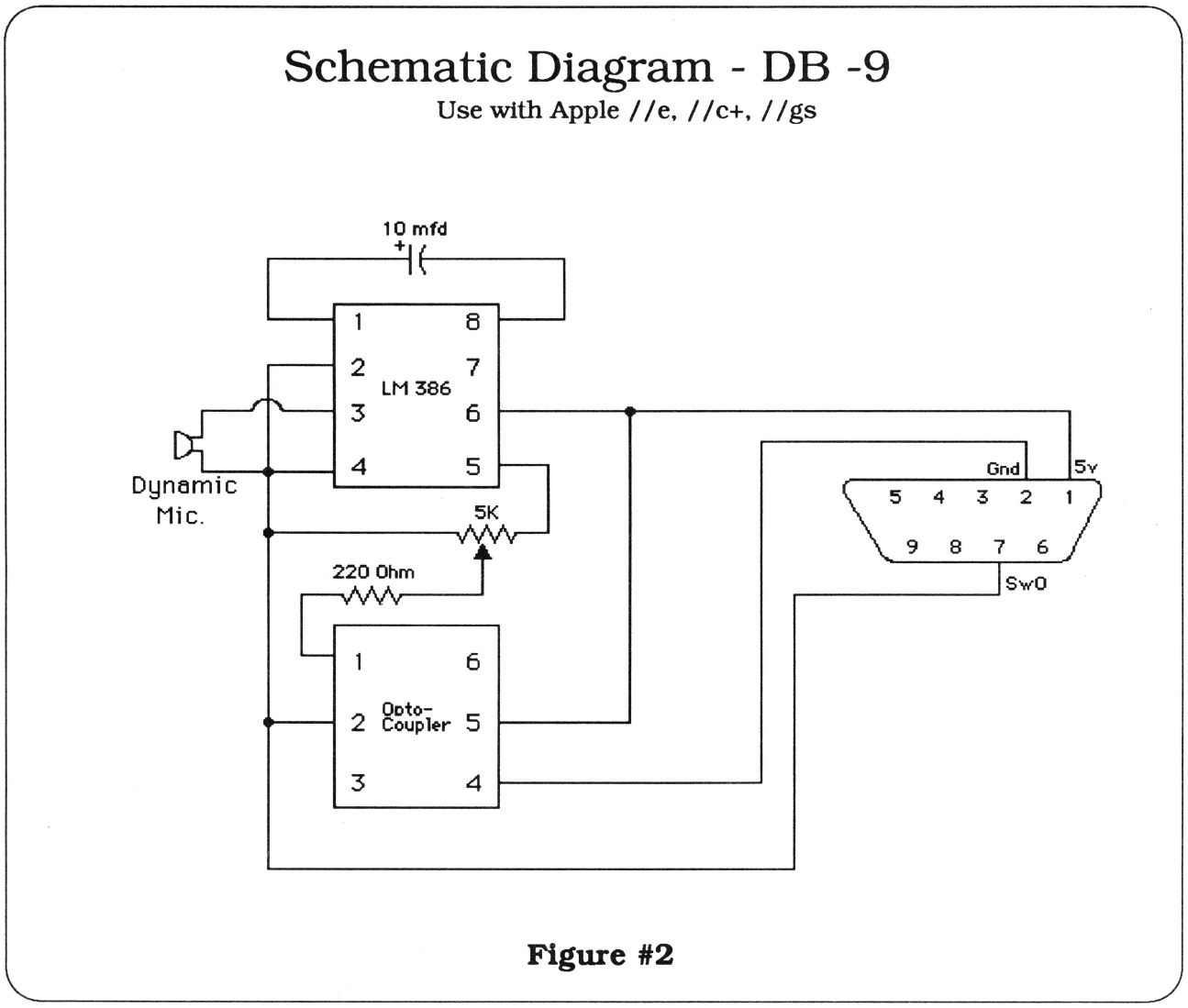


Figure #2

lar to but in contact with my lips and speaking past (not directly into) it. This way it's always in the same place with respect to the volume and sounds your making. (A boom mic would help here.) If the distance between the mic and your lips varies much at all between the training and recognition phases,][-Ears begins to see different waveforms generated by the hardware causing recognition errors.

When adjusting the pot, turn it until the asterisks appear during words and particularly the "s" sound (try hissing like a snake), but don't appear when you're breathing in or out. If many asterisks appear during your breathing this indicates that the bias level (pot) is too high. Turn the knob so that fewer asterisks appear. This adjustment can be a bit tricky, but it's not hard once you get the hang of it. Note that you'll probably need to readjust this setting whenever you change opto-couplers or microphones.

Also, once this adjustment is made, leave it for the duration of your session. If you move the knob during recognition,][-Ears will begin to misidentify your words most of the time. The only solution is to readjust the level using the two line adjustment program, then retrain the system from scratch. This can be a bit of a nuisance. For turnkey systems where the user is not technically minded, the best bet would be to build the interface box with the potentiometer inside, accessible only with a screwdriver through a hole drilled in the box. This way you could set the adjustment once and forget it. However, since this is a hardware hacking column, I

figured you'd want the adjustment right out where you can get at it!

If you can get your hardware to respond as described above, then it's working properly. If not, double check your connections, the chip orientation, and for shorts. Also, some opto-couplers work better than others. Since you got 3 in the Radio Shack package, try one of the others. My particular package had a chip number TIL 119 which did not work as well as the other two.

Entering the Software

Next, boot up your assembler, then type in and assemble the source code called EARS.S. (See Listing 1). Alternately, you can enter just the machine code directly from the monitor. Save it on disk with the command:

BSAVE EARS.OBJ,A$6000,L496

At this point, assuming that there are no hardware or software bugs, the system is ready to go. If you'd like a simple Applesoft demonstration of one way to use speech recognition in your own programs, type in the Listing 2 called "RECOG.DEMO"

RECOG.DEMO: A Demonstration Program

RECOG.DEMO shows two elementary ways to use speech recognition in your programs: controlling

Parts List for 16 Pin DIP Version

Item	Catalog #	Price
Box w/ circuit board	270-283	$3.99
LM386 amplifier chip	276-1731	$1.09
Opto-Couplers (Pkg of 3)	276-139	$1.98
Chip Sockets (Pkg of 2)	276-1995	$ .59
5K potentiometer	275-1714	$1.09
220 ohm resistor (1/4 watt)	271-1313	$ .39
10 microfarad capacitor	272-1025	$ .59
Knob (Pkg of 2)	274-402	$1.19
Microphone mini-jack (Pkg of 2)	274-248	$1.69
Wire (2 conductor w/ shield - 3 ft)	278-1276	$3.29
16 Pin DIP Header*	N/A	$ .65
	Total	$16.54

* The 16 Pin DIP Header is not made by Radio Shack anymore. Instead, it is part #16 HP available from Jameco Electronics, 1355 Shoreway Road, Belmont, CA 94002 (415) 592-8097. It is also available from many other sources.

Figure #3

Parts List for DB-9 Version

Item	Catalog #	Price
Box w/ circuit board	270-283	$3.99
LM386 amplifier chip	276-1731	$1.09
Opto-Couplers (Pkg of 3)	276-139	$1.98
Chip Sockets (Pkg of 2)	276-1995	$ .59
5K potentiometer	275-1714	$1.09
220 ohm resistor (1/4 watt)	271-1313	$ .39
10 microfarad capacitor	272-1025	$ .59
Knob (Pkg of 2)	274-402	$1.19
Microphone mini-jack (Pkg of 2)	274-248	$1.69
Wire (2 conductor w/ shield - 3 ft)	278-1276	$3.29
DB-9 connector (male)	276-1537B	$ .99
	Total	$16.88

Figure #4

program flow, and item selection from a menu. Many other things are possible, I just chose these because they were the first two things that came to mind!

When you run RECOG.DEMO, the first few screens illustrate the idea of controlling program flow. A screen of text information is presented along with a prompt at the bottom of the screen which says "Say 'OK' to continue..." At this point,][-Ears is waiting for any and all microphone input. When you say "OK", the microphone picks it up, program flow returns to Applesoft, and the next screen of information is presented.

This is a more elegant solution to the familiar "Press <RETURN> to continue..." prompt used so much to control program flow. You'll note that at this point in the program,][-Ears is yet to be trained. Because of this, it is not really recognizing the word OK. Instead, it is just responding to any sound at all. Rapping the mic on the desk, or saying a nonsense word would work just as well as saying OK. This does not have to be the case, however. If you like,][-Ears can be trained to understand the word "OK" and then respond only if you say it. This would prevent the program from responding to background noise or inadvertent triggers (talking to someone else, dropping the mic, etc.)

Following the information screens, you train the system to understand your voice. At this point you are asked to repeat the numbers 1 through 6 and the word "return" ten times. Say the words naturally, but remember how you say them. The biggest problem people have with this system is that they say the words differently during training than they do when they expect][-Ears to identify their speech in a program.

When the training is finished you're presented with a standard menu. The numbers you just trained the system to understand match up with the numbered items on the menu. To highlight a menu item, simply say its number. To execute a highlighted item, say "return". This is essentially a classic AppleWorks-

style menu implemented using exclusively voice control. The RECOG.DEMO program is not meant to be a polished example of what can be done. It merely shows a few ways to use the EARS machine language subroutines and voice control in a typical Applesoft application.

Theory of Operation

][-Ears thinks of your speech not as letters and sounds but as waveforms and frequencies. These waveforms change drastically depending on what sound you're making with your mouth. The different classifications of sounds all have a different waveform. For example, fricative sounds such as "F" or "S" have a random, high frequency waveform. Plosive sounds like "P", "T", "K", and many times "CH" have a short burst of high frequencies followed by a longer period of lower frequencies. Each sound has an identifying frequency response fingerprint.

Words usually consist of more than one sound. Over the course of an entire word, the frequencies present will change according to the sound currently being formed. To recognize a word by its frequency response, we need to know what frequencies are present at several points along the word. The more points where we identify the frequency spectrum, the better. The objective is to create a map of the frequencies present at several points along the word that we can compare to frequency maps of other words. If the maps match, we've found the word. At least we hope we've found the word!

][-Ears works by recording all sounds coming in the mic for about 3/4 of a second. It then analyzes the resulting waveforms for frequency content at four equally spaced points along the recording. Next, it condenses each frequency analysis down to 8 bytes, resulting in a total of only 32 bytes of identifying information for each word. This last step drastically cuts down the information required to store a word's identifying information. It also facilitates and speeds up comparisons.

How It Works - Hardware

The hardware in this system is simple because

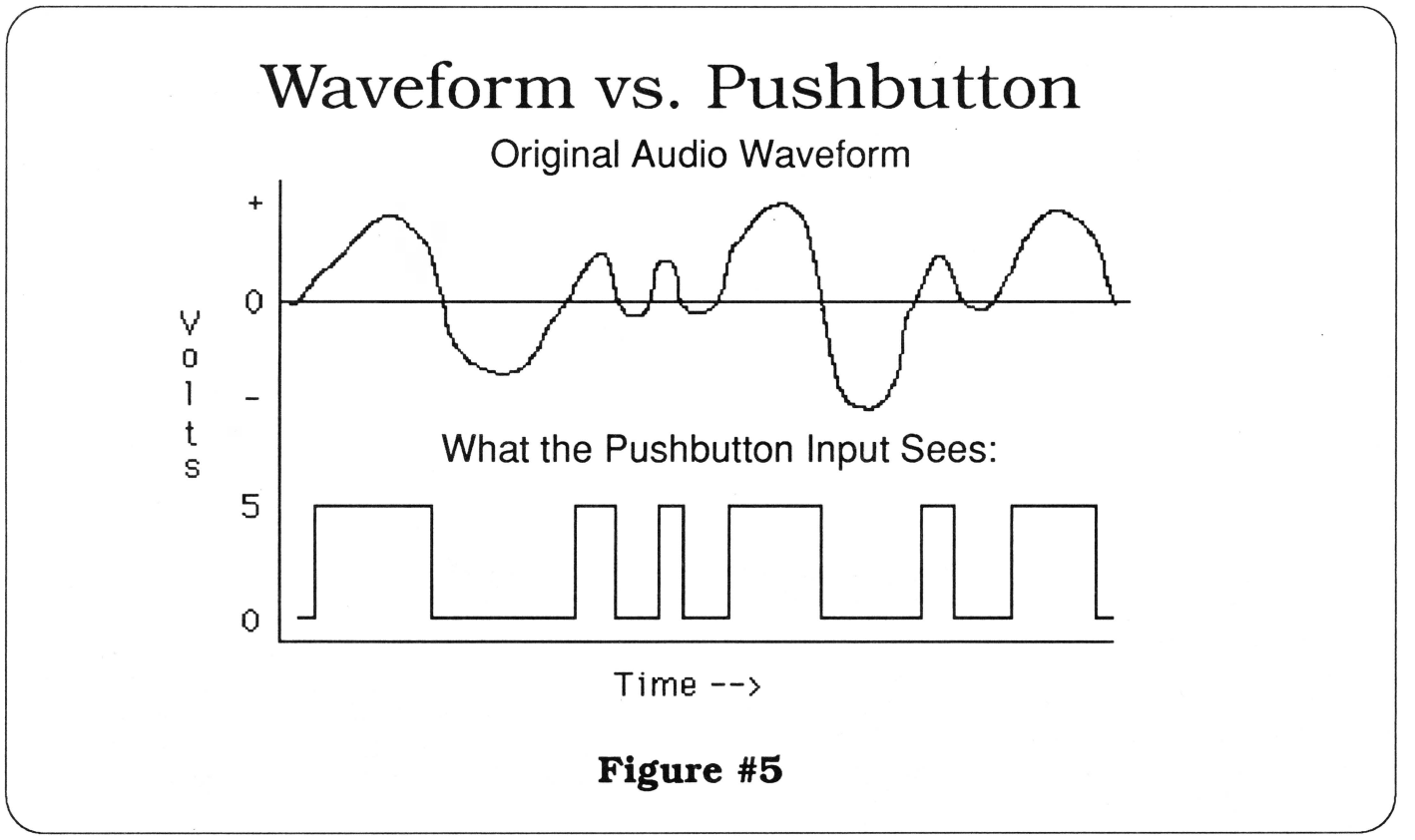


Figure #5

almost all the work is done in software. The only function the hardware has is to "push" the gameport pushbutton in sync with the waveform your voice creates as you say the words.

The first chip, an LM386, is a simple audio amplifier. (If you really want to impress someone, just tell them][-Ears uses a '386 chip!) The output from a microphone is very small, and nowhere near what we need. The LM386 boosts the signal from the mic up to usable levels. The output from the LM386 is used to drive an LED (light emitting diode) so that the LED brightness varies with the waveform of your voice.

The second chip, an opto-coupler, does most of the hardware work. Despite it's fancy name, this device contains just the familiar LED and a phototransistor linked optically in the same package. When we apply our waveform voltage from the LM386 to the LED side of the opto-coupler, the LED brightness follows the waveform. The phototransistor sees this light and begins to conduct electricity according to the brightness of the light it sees.

In essence, the phototransistor acts as a switch. When the waveform swings positive, the LED lights up causing the transistor switch to turn on and connect the pushbutton input to 5 volts. The Apple sees 5 volts on the pushbutton input and knows a pushbutton has been "pushed." When the wave goes negative the transistor switch opens disconnecting the voltage, thereby releasing the button.][-Ears uses the switching action of the phototransistor to alternately connect and disconnect the pushbutton input to 5 volts. (See Fig. 5) Your Apple is unaware that the circuitry is doing the button pushing. All it knows is that something is pushing the button awfully fast - hundreds of times a second!

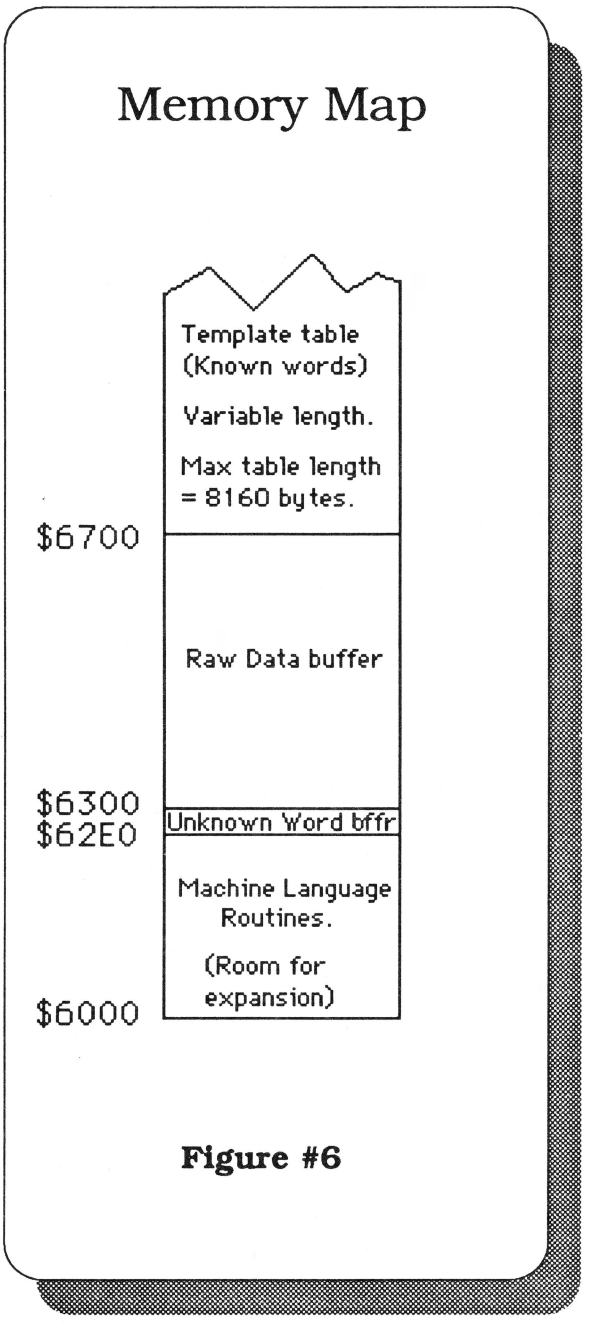


Figure #6

How It Works - The Software

The][-Ears software has 2 subroutines. RCRD handles all incoming words. Words are either stored in the table (learned) or stored in a buffer to be compared against those already stored in the table (recognized). FIND does the speech recognition by searching through all previously learned words looking for a match.

Each word is condensed into a template. Each template has a number. If you record a word more than once (highly recommended), then each word can have more than one template. To reference a template in the][-Ears routines you pass it's number in location 255 (hex $FF) which I've called WORDNUM. A template's number can be anything from 0 to 254, and is assigned by the host program. After recognizing a word, FIND returns the number of the word it found in WORDNUM and a the word's score in SCORELO, and SCOREHI (253 and 254, respectively) which form the low and high byte of a 16 bit number. Before we discuss these routines, let's look at][-Ears memory usage.

As currently assembled,][-Ears resides at $6000 which is just above HIRES page 2. (See Fig. 6 for memory map.) The machine language routines, input buffer and overhead are 1792 bytes long total. Immediately following this at $6700 in memory is the template table. Template data for each word to be identified is placed at a location in this table according to a number you pass to the record routine in $FF (WORDNUM). Since][-Ears numbers words from 0 to 254, and each template is 32 bytes long, the longest this table can be is 8160 bytes. More likely, you'll have only 60 or 70 templates in the table resulting in a table length somewhere around 2K (2240 bytes). With 70 templates in the table, the total length for the machine language and template table portion of RECOG.DEMO is 4288 bytes. (We'll talk about why there are 70 templates in the table when][-Ears has to discriminate between only 7 words later.)

The Machine Language Routines

The record routine (RCRD) is the heart of the system. This is the routine that records raw data, analyzes it for frequency content, and finally condenses this information down to a 32 byte template. This template is stored in a table of templates at an address dependent on what number you pass in WORDNUM.

The number passed in WORDNUM acts both as a flag and as the number of the table location to store the resulting data at. If you pass 255 (hex $FF - acts as a flag), RCRD knows that the incoming word is to be identified, not stored as an entry in the table, and places the final data at $6300 which is the input location. If you pass any other number (0 to 254), RCRD knows that you want it to "learn" this word and place the final data in the template table which starts at $6700.

When RCRD is first called it clears the 32 locations that will be filled with new template data to zero, then waits in a loop until it hears the first sound. After detecting the first sound it begins to fill the raw data buffer with samples of the incoming audio signal. RECOG.DEMO relies on the fact that RCRD waits in this loop when using RCRD to control program flow.

The raw data buffer, located at $6300, is 1024 bytes long. When recording, we "take pictures" (samples) of the pushbutton input 8192 times. Since it takes about 3/4 of a second to take all 8192 samples, we have a sampling rate of roughly 10,000 samples/second. According to Nyquist's theorem, this is high enough to record frequencies as high as 5000 Hz which is adequate for our purposes.

"Since it takes about 3/4 of a second to take all 8192 samples, we have a sampling rate of roughly 10,000 samples/second."

The pushbutton is either on or off at any one time, so we can represent its state with one bit. If the button is pushed at the instant RCRD samples the sound wave, it stores a one in the raw data buffer. If not pushed, RCRD stores a zero. Each byte will hold 8 separate bits so we can pack 8 samples in one byte. This is how 8192 samples will fit in 1024 bytes.

The hardware's job is to push the button in sync with the incoming waveform. This is so that when RCRD is done sampling, the raw data buffer will contain a digital representation of the waveform in the form of a series of 1's and 0's. (See Fig. 7) Since we know that one complete wave (cycle) consists of both a positive and a negative swing of the wave, and we also know that the samples come at regular intervals (about 1 every 10,000th of a second), we can calculate the relative length of the wave by counting the number of samples in one whole cycle.

A complete cycle consists of a string of 1 bits followed by a string of 0 bits. When the next string of 1 bits starts we know this particular cycle is finished. Obviously, short cycles will have smaller counts and long cycles will have larger counts. (See Fig. 8) Frequency and cycle length are directly related. If we know the length of a cycle, we also know its frequency.

Frequency Analysis

The next step is to sort the cycle length (frequency) into one of 8 frequency ranges. Once we have the

Call Box™ TPS

The Toolbox Programming System

Finally... a BASIC you can use!

All the features that once made Applesoft the language of choice among most users are still valid today. However, the increased functionality of today's Apple II requires access beyond the capabilites available in regular Applesoft. The Call Box TPS provides you with the "*missing link*" necessary for programming the advanced features of the IIgs while maintaining the simplicity and feel of good old Applesoft.

✱ Immediate mode access to the tools... commands are directly executable from the keyboard! ✱ The most common tool functions are automated by simple call structures! ✱ No Assembly, Linking or Compiling is required... the ideal prototyping language! ✱ Most GS/OS and ProDOS 8 calls are available at the same time! ✱ Capable of fine Machine Code like control using specialized commands! ✱ Totally Memory Managed and compatible with NDA's, CDA's and initialization code! ✱ Directly launchable from programs like the Finder™ or HyperLaunch™! ✱ Uses templates generated by the Call Box WYSIWYG Editors such as Windows, Dialogs Menus, Icons and Cursors!

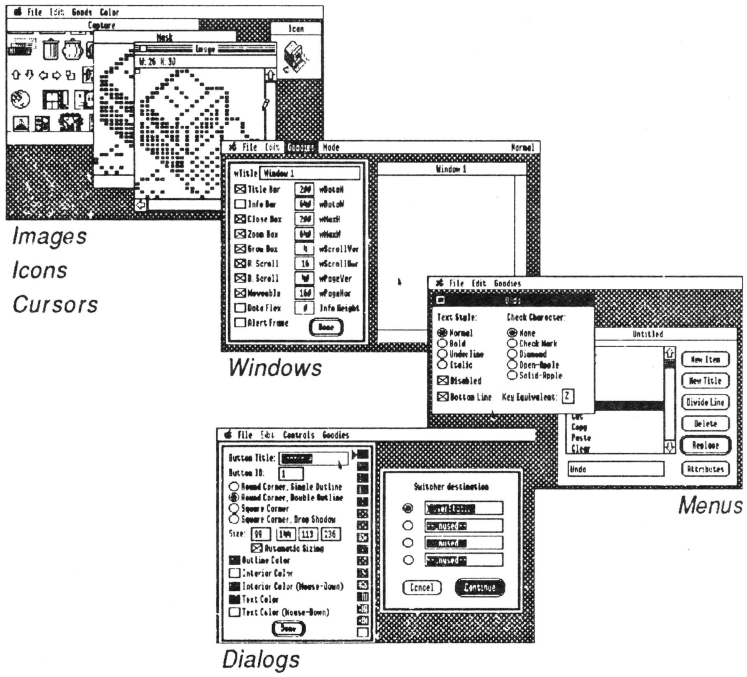
Creating a **Call Box BASIC** program is a simple 2 step process. Step 1 involves using the Call Box WYSIWYG editors to design any graphic entity needed by your program such as Windows, Menu, Dialogs, Icons, Cursors or Pixel Images. Step 2 consists of incorporating your entities into a BASIC program using specialized calls provided by the BASIC driver.

Currently, the system has four WYSIWYG (What You See Is What You Get) Editors which produce the various graphic entities used by the Call Box BASIC Driver such as Windows, Dialogs, Menus, Icons, Cursors and Pixel Images. Just "point and click" to compose entites exactly to your liking without any of the guesswork or number crunching associated with the "paper and pencil" method.

You want to change an items color? Just click it... You want to add a radio button? Just click it. When everything is the way you want it, just save it!

The entities created by the editors are not limited to Call Box BASIC programs; the editors also produce APW-ORCA/M™ source code, object code and relocatable resource fork data allowing other languages such as Assembly, C or Pascal to enjoy the full benefits of object oriented programming. Once you have created your entities, you can incorporate them into your program which greatly reduces the setup and overhead usually associated with these structures.

The **Call Box TPS** comes on 3 - 3.5 inch disks and has a 130+ page hard cover ring binder manual. The disks include demos,



samples and utilities to ease the task of creating a program. The system is installable on hard drives or can be run from as little as 1 - 3.5 inch disk drive. The recommended memory for this system is 1 Megabyte (*minimum*), however 768K is enough for most applications.

Continued support for this system is available for a nominal annual membership fee thru the **Call Box Programmers Association (C.B.P.A.)**. This association supplies you with the latest tech notes, sample code disks, software and manual upgrades plus a programmers hot-line number to help you with those "tricky" procedures that make your Apple IIgs do amazing things! You will also receive newsletters and information on other So What Software products as well.

The **Call Box TPS** is the total programming environment for the Applesoft BASIC programmer using the Apple IIgs, and ***YES!***... Finally there is a **BASIC** you can use!

Call Box TPS.....$99.00

So What Software™

10221 Slater Ave. Suite 103 Fountain Valley, CA. 92708

(714) 964-4298 VISA/Mastercard accepted

length of the wave we can figure out which of the frequency bins it belongs in through a series of comparisons to threshold values. (The thresholds are the compare values found in the BINSORT routine at line 153 and following in the EARS.S listing.) This step is where the actual frequency analysis takes place.

Frequency analysis is important to][-Ears because it relies on the fact that the frequencies present at particular points along a word distinguish it from other words. Most of the information that distinguishes one word from another is found in the higher (smaller numbered) frequencies. For this reason, the thresholds for the higher frequencies are very close together while lower frequencies all get lumped together.

Generally, the threshold frequencies were chosen to detect the formant frequencies generated by the resonating of the various cavities in your head, mouth and throat. (Analyzing the formant content of a particular sound is a common way to do speech recognition.) By setting the thresholds to detect the formant frequencies we do a crude formant analysis and avoid having to go through the usual Fourier transforms with all their attendant math. If you find a series of threshold values that work better than these, let me know!

Recall that][-Ears does a frequency analysis at 4 points along a word time-wise. Remember also that the raw data for a word is 1024 bytes. Obviously, since the data was taken sequentially, the data at the beginning of the buffer was taken at the start of the word and the data at the end of the buffer came from the end. By dividing the raw data buffer into 4 equal sections of 256 bytes each (I call them "time slices"), we end up dividing the word into 4 different parts time-wise.

The four separate frequency analyses are accomplished by sorting the cycle lengths found in each time slice into their own set of 8 frequency bins.

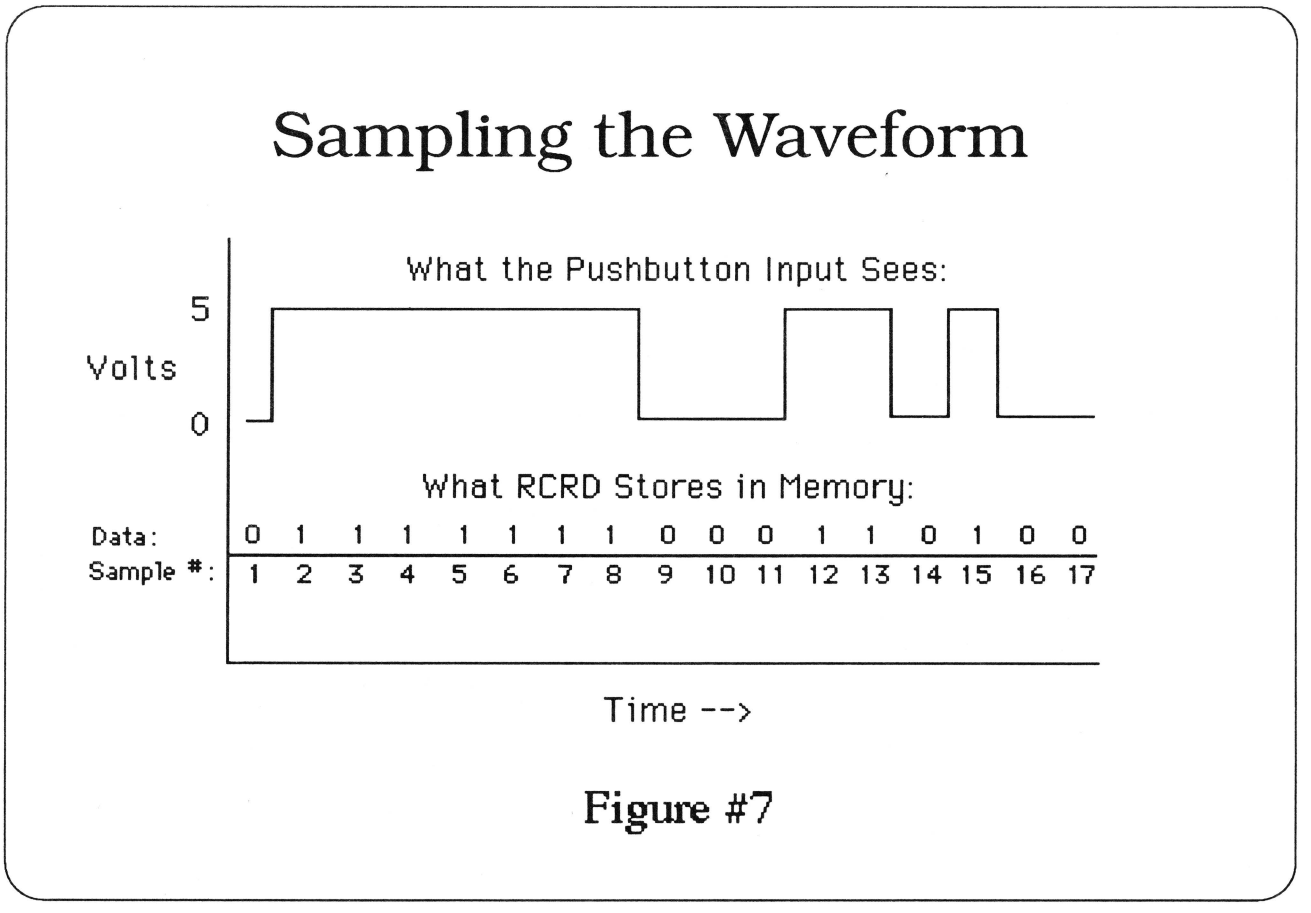


Figure #7

Measuring Wavelengths

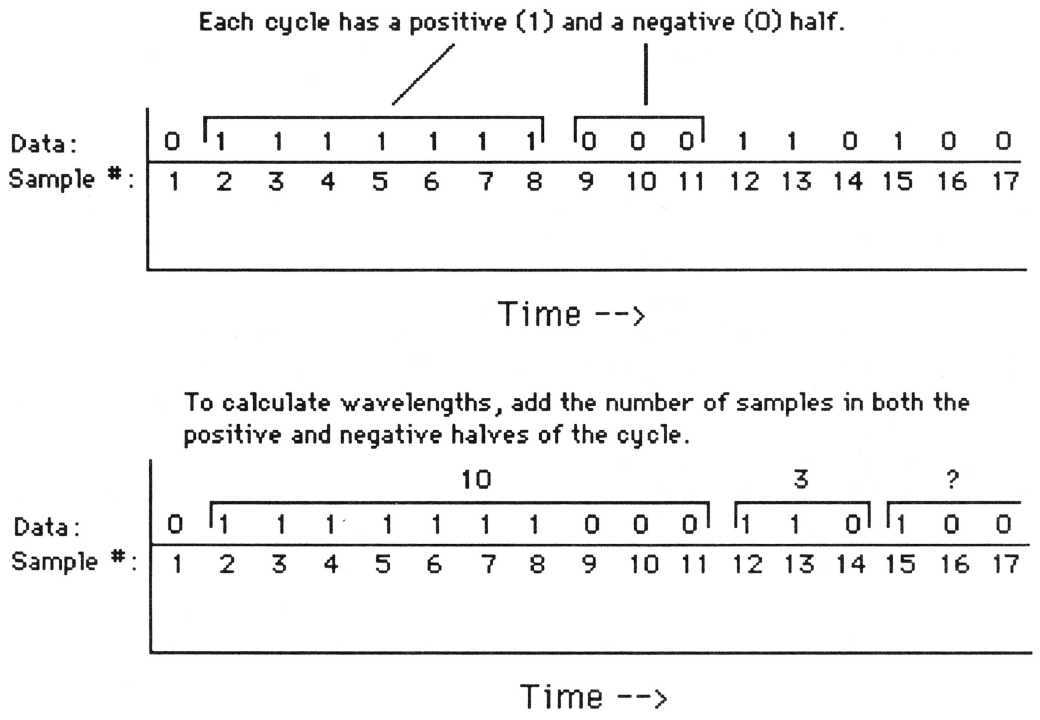


Figure #8

(Each time slice has 2048 samples contained in 256 bytes). The first time slice gets sorted into the first 8 frequency bins, the second time slice into the second into the second 8 frequency bins, and so on, until all four time slices have been sorted into their respective bins. (See Fig. 9) This results in 32 bytes of data which we'll call a "template."

RCRD stores the resulting template data in a table which starts at $6700. The exact location depends on what value N you passed to the routine in WORDNUM. Since each word results in 32 bytes of data, each entry in the table is an offset of N*32 from the base location of $6700. If you passed a value of 5 in WORDNUM, for example, the template data would be stored in the 32 bytes starting at $67A0. (5x32=160 which is hex $A0, and obviously, $6700+$A0=$67A0)

The Recognition Algorithm

The actual recognition of speech is accomplished by the FIND routine. It compares the template of the unknown word found at $6300 to all the templates in the table at $6700. FIND can search up to 255 templates. Because searching all 255 templates is unnecessary when you may only have 70 entries in the table, FIND allows you to specify how many templates you would like it to search. Pass the number of the highest table entry you want searched in WORDNUM before calling FIND. FIND works backwards from the number in WORDNUM to template 0 in its search. Any higher numbered templates will not be searched. This makes it possible to have several different vocabularies in memory at once if you plan their locations in the table carefully.

The mechanism FIND uses to recognize a word is surprisingly simple: subtraction. To subtract means to find the difference between 2 numbers. Each word is represented by a 32 byte template, and it is a simple matter to subtract one word's template from another's to find out how different they are. The first byte in the template for word "A" is subtracted from the first byte in the template for word "B", the second byte of "A" is subtracted from the second byte of "B" and so on until all bytes have been subtracted. During the subtraction process it is not important whether the result of the subtraction is positive or negative. We only want to know the magnitude of the difference between the two numbers.

FIND adds the result of each subtraction to the previous total to form a cumulative score (total difference) for the word which is passed back to the calling program. The bigger the score, the more different the two templates are. A perfect match results in a score of zero. Speech recognition becomes a simple matter of finding which template in the table has the smallest score when compared to the unknown word's template. This method is quite fast:][-Ears can search through 255 templates in roughly 1/2 second!

Fast Talkers Beware

The speed of speech can vary greatly. We may say a word quickly once, then rather slowly the next time. For][-Ears, words said quickly may have all sorts of data for the word in the first 3 slices, for example, and nothing in the 4th slice. This would create a large error count when subtracting templates during recognition. Recognition errors would result.

For this reason,][-Ears weights the various time slices in terms of their contribution to the final error total. If the error to be added to the total comes from one of the first 2 slices, then the full value of the error is added to the total. If the error comes from the 3rd slice, it's divided in half before being added to the total so it has only a 50% weighting. If it's from the last slice, it's divided yet again so that it has only a 25% weight in the final total. This minimizes the effects of speed changes when you say words because large differences in the last 2 slices have less effect on the final score than they would have otherwise.

The number of template with the lowest score is passed back to the host program in WORDNUM and its score (which is used as the confidence level) is placed in SCORELO and SCOREHI. Whereas FIND identifies the template with the lowest score, this template may or may not represent the right word. The final decision is made by the host program based on exactly how different the words are i.e. how high the score is. After all, if the best match has a relatively high score (greater than about 60 or 70), chances are we have not found the unknown word. If the difference is relatively low (20-30) it's quite likely that we have found the word. To sum up, FIND returns with the best match, but not necessarily the correct word.

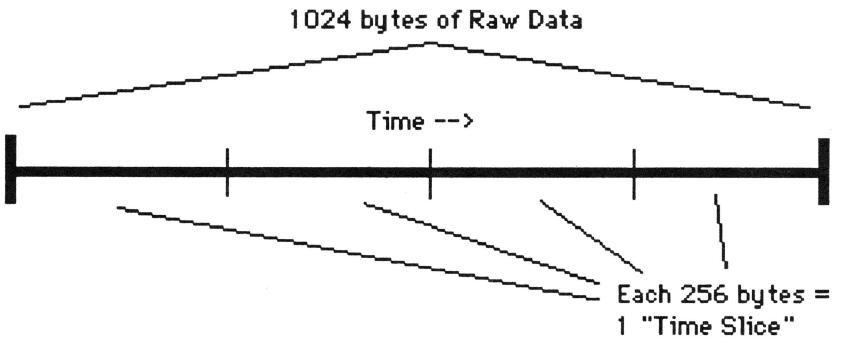
Problems with Recognition

No matter how hard you try, you'll never be able to say a word exactly the same way twice. This causes][-Ears problems because it's trying to match two words that aren't exactly the same. For example, the template in the table may reflect the fact that your vocal chords vibrated 439 times over the course of the word when you trained the system. However, when you wanted EARS to recognize the word, your vocal chords vibrated 445 times and you said the first syllable slightly stronger.

One way to help][-EARS deal with these differences is by entering multiple templates of the same word in the table when you train the system. This captures several different ways you say one of the target words. Then, no matter how you actually say the word, one of the templates will match during recognition. Clearly, the more templates you have for a particular word, the better off you are. The trade-off is that sooner or later, the user must train the system by saying the words many times. Also, the more templates you devote to each word, the fewer different words you'll have room for.

Using multiple templates doesn't solve all problems, though. Some of the recognition errors stem from how people say the words during the training phase. RECOG.DEMO asks you to train the system with 7

Raw Data vs. Condensed Form



High Frequencies	0-3
	4
	5
	6
	7-8
	9-10
	11-12
Low Frequencies	13+

Within each time slice, cycle lengths are sorted into 8 different frequency bands.

Each frequency band is represented by one byte which is the total count of all the cycles found within that band of cycle lengths.

A word is stored in memory as a string of 32 bytes. While shown here as a 2 dimensional array for visual clarity, the bytes are actually sequential in memory: first all the bins for slice one, then all those for slice two, etc.

Time Slices:

Frequency Bins	1	2	3	4
1	$C8	$F9	$D7	$E3
2	$D6	$E4	$CA	$DD
3	$BD	$CC	$13	$03
4	$43	$5D	$24	$22
5	$64	$7A	$10	$29
6	$2A	$3D	$11	$09
7	$14	$D9	$23	$15
8	$04	$12	$09	$08

Figure #9

different words, each of which is said 10 times which amounts to 70 words. When saying these same 7 words over and over again, people's voice inflection can get either "sing-songy" or like a monotone. As a result,][-Ears records templates that don't represent how the person normally says the words. Later, during the recognition phase, the person may say the words more naturally.][-Ears then compares the incoming natural pronunciations against the monotone ones stored in the template table and either mis-identifies words or doesn't understand what they're saying.

"The thing to remember is that the more words][-Ears must identify, the more mistakes it makes."

The obvious solution is to concentrate on saying the words naturally - just like you will say them when they need to be identified later. Another tip is to hold the microphone in the same place during both the training and recognition modes. Observing both of these suggestions will ensure a more accurate training phase and fewer recognition errors later on.

Using][-EARS in Your Own Programs

Adding speech recognition to your own programs is easy with][-Ears. First, BLOAD the routines into memory with the Applesoft command:

```
PRINT CHR$(4);"BLOAD EARS.OBJ"
```

Next, train the system. In other words, record the templates that unknown words will be compared to. First, poke the number of the word to be learned in WORDNUM then call RCRD. This word number can be any value from 0 to 254, but remember to keep track of which number represents which word. For greatest accuracy, make at least 5 recordings of the same word as discussed before using 5 different word numbers.

Once all the words have been learned, the system is ready to recognize speech. To identify a word it must first be recorded and stored in a special buffer.][-Ears takes care of all this when you poke $FF in WORDNUM. This shows RCRD that the next word is to be identified, not stored in the template table. Next call RCRD and speak the word to be recognized. Finally, poke WORDNUM with the highest word in the template table you want searched and call FIND. That's all there is to it. FIND returns the number of the best match in WORDNUM, and the difference between the two templates in SCORELO and SCOREHI (locations 253 and 254 respectively).

What's the Score?

To decide whether you have actually found the word, get the word's score with a statement such as:

```
SCRE= 256 * PEEK(254) + PEEK(253)
```

where SCRE will hold the score. Compare the score to a threshold level with a statement like:

```
IF SCRE > 60 THEN GOSUB 1000
```

where line 1000 begins the routine to handle an incorrectly identified word. In this case, if the score is below 60 the program assumes we have found the correct word. If it's 60 or above, then it assumes we have a false match and calls the error routine which probably asks the user to try again. Obviously, the lower the threshold level is, the closer a match it will have to be to avoid calling the error code.

If FIND has truly found the word, get the word number with a statement like:

```
WRD=PEEK(255)
```

where WRD will hold the word number.

Applications

Now that your Apple understands speech, what can

you do with it? Anything that requires a limited vocabulary is a candidate for conversion to voice control. True/False, Yes/no, High/Low, and Up/Down pairs of responses all work very well with][-Ears. Menus with less than 10 or so possible choices also work well. Numerical input is possible one digit at a time.][-Ears can handle all ten digits (0 through 9) plus the decimal point very easily. With 255 locations available for templates, it is possible to train it to understand the entire alphabet using 9 templates per letter.

The thing to remember is that the more words][-Ears must identify, the more mistakes it makes. There are ways around this. One is to ask for verbal confirmation after a choice has been made. In the RECOG.DEMO program you'll notice that saying a number didn't activate a selection, it only moved the highlight bar to the item corresponding to that number. This allows you to say the number again if][-Ears makes an error. Activation occurs only when you say "return". In effect, this is a kind of confirmation of your selection.

Other ways to minimize errors include using fewer words, and words that sound quite different such as the yes/no or up/down pairs. The AppleWorks menu could be implemented using the words up, down, and return, for example. Up and down would move the highlight bar and return would activate the highlighted selection.

Experimentation with all sorts of esoteric things is possible. How about identifying someone by the sound of their voice? If you have a speech synthesizer, interaction with the computer could be totally verbal: you speak to the computer and it speaks back!

Since you have total control of the software, it is possible to incorporate artificial intelligence in the system as well. Suppose during the training phase, you programmed the computer to learn from its mistakes. If it mis-identifies a word, you could either add the template that fooled][-Ears to the table, or use it to change the existing templates somehow. This way the system would learn as you go and make fewer and fewer errors.

Modifications and Improvements

Apple][‘s through the //e have a cassette port that functions very much like][-Ears' hardware. Theoretically, it should be possible to make][-Ears software work with that port. Radio Shack makes a small battery powered amplifier (cat. #277-1008) that could be used to amplify the microphone. Output from the amp's external speaker jack could then be fed into the cassette input on the Apple. Because the cassette port electronics may differ in response to what the][-Ears software expects, you may need to tweak the threshold values in the BINSORT machine language subroutine. (BINSORT is at line 153 of the source listing.) I haven't tried this but in theory it should work.

On the other hand, you can use][-Ears hardware to simulate the cassette port hardware no longer found on later Apples. It is now possible to use all those speech digitization programs that used the cassette port for input. I even remember seeing an article that used the cassette ports for a crude (but inexpensive!) local area network.

It is possible to account for the higher clock rates afforded by the GS and accelerator boards. By adjusting the delay to a larger value in line 62 of the assembly, it should be no problem to make][-Ears work on a computer with a clock rate higher than the standard 1 MHz Apple clock.

Conclusion

][-Ears offers an inexpensive alternative to keyboard entry for any Apple][computer. It is easy to use in your own programs and doesn't take much memory, particularly if you need to identify just a few words. It's also fun! Who knows what you can do with your Apple now that your computer has ears!

Late Breaking News:

Ariel Publishing, Inc. acquires FAX capabilities. Our FAX phone number is:

(509) 689-3136

Listing 1 - RECOG.DEMO

```
 100  REM RECOG.DEMO
 110  REM  C 1990 BY
 120  REM  DAVID GAUGER II
 130  GOSUB 1610: REM INIT SYSTEM
 140  GOSUB 1020: REM LEARN WORDS
 150  REM ** MAIN MENU **
 160  HOME
 170  VTAB 2: HTAB 11
 180  INVERSE : PRINT "SPEECH INPUT MENU":
NORMAL
 190  FOR X = 0 TO 5: GOSUB 840: NEXT X
 200  VTAB 19: HTAB 5
 210  PRINT "SAY THE ITEM NUMBER TO SELECT,"
 220  HTAB 5
 230  PRINT "'RETURN' TO EXECUTE."
 240 X = 0: GOSUB 890
 250  GOSUB 940: REM GET A WORD
 260  IF ER > ELEVEL THEN  GOSUB 780: GOTO
250
 270 WRD =  INT (WRD / MAX)
 280  REM  ** MOVE HIGHLIGHT BAR
 290  IF WRD + 1 < 7 THEN  GOSUB 840:X = WRD:
GOSUB 890: GOTO 250
 300  REM  ** IF RETURN, DO ITEM
 310  ON X + 1 GOTO 330,340,420,480,500,620
 320  REM ** DO BOX/ERASE BOX
 330  INVERSE : GOTO 350
 340  NORMAL
 350  VTAB 1: HTAB 1: FOR Y = 1 TO 39
 360  PRINT " ";: NEXT
 370  FOR Y = 2 TO 20
 380  VTAB Y: HTAB 1: PRINT " ";: HTAB 39:
PRINT " ";: NEXT Y
 390  VTAB 21: HTAB 1: FOR Y = 1 TO 39: PRINT
" ";: NEXT Y
 400  GOTO 250
 410  REM ** DO CATALOG
 420  PRINT  CHR$ (4);"CATALOG"
 430  PRINT : PRINT : PRINT
 440  VTAB 23: HTAB 5
 450  PRINT "SAY 'OK' TO CONTINUE...";
 460  GOSUB 940: GOTO 160
 470  REM  ** DO BEEP SPEAKER
 480  FOR Y = 1 TO 3: PRINT  CHR$ (7): NEXT
Y: GOTO 250
 490  REM  ** DO HIRES DEMO
 500  TEXT : HOME : HGR : HCOLOR= 3
 510 H1 = 0:V1 = 0:H2 = 279:V2 = 0
 520 H3 = 279:V3 = 159:H4 = 0:V4 = 159
 530  FOR X = 0 TO 10: GOSUB 590: NEXT
 540  FOR X = 20 TO 30: GOSUB 590: NEXT
 550  FOR X = 40 TO 50: GOSUB 590: NEXT
 560  FOR X = 60 TO 70: GOSUB 590: NEXT
 570  GOSUB 1570: TEXT : GOTO 160
 580  FOR X = 1 TO 30
 590  HPLOT H1 + X,V1 + X TO H2 - X,V2 + X TO
H3 - X,V3 - X TO H4 + X,V4 - X TO H1 + X,V1 +
X
 600  RETURN
 610  REM  ** DO QUIT
 620  HOME
 630 X = 0
 640  VTAB 18: HTAB 3
 650  PRINT "SAY THE ITEM NUMBER TO SELECT,"
 660  HTAB 3
 670  PRINT "'RETURN' TO EXECUTE."
 680  IF X = 0 THEN  VTAB 10: HTAB 8: INVERSE
: PRINT QUIT$(0): NORMAL : VTAB 12: HTAB 8:
PRINT QUIT$(1)
 690  IF X = 1 THEN  VTAB 10: HTAB 8: PRINT
QUIT$(0): VTAB 12: HTAB 8: INVERSE : PRINT
QUIT$(1)
 700  NORMAL : GOSUB 940
 710  IF ER > ELEVEL THEN  GOSUB 780: GOTO
620
 720  IF WRD < 6 THEN X = 0: GOTO 680
 730  IF WRD > 9 AND WRD < 20 THEN X = 1:
GOTO 680
 740  IF WRD > 35 AND X = 1 THEN 760
 750  GOTO 160
 760  HOME : END
 770  REM ** DO ERROR
 780  PRINT  CHR$ (7);: VTAB 22: HTAB 5:
INVERSE : PRINT "COULD NOT UNDERSTAND - TRY
AGAIN"
 790  NORMAL
 800  FOR Z = 1 TO 500
 810  NEXT Z
 820  VTAB 22: CALL  - 958: RETURN
 830  REM ** DO NORMAL MENU ITEM
 840  VTAB 5 + 2 * X: HTAB 8
 850  NORMAL
 860  PRINT ARRAY$(X)
 870  RETURN
 880  REM ** DO INVERSE MENU ITEM
 890  VTAB 5 + 2 * X: HTAB 8
 900  INVERSE
 910  PRINT ARRAY$(X): NORMAL
 920  RETURN
 930  REM ** DO RECOGNITION
 940  POKE WNUM,255: REM  SET TO LISTEN
 950  CALL RCRD: REM  GET WORD
 960  POKE WNUM,MAX * 7: REM HIGHEST WORD TO
SEARCH
 970  CALL FIND: REM  FIND WORD
 980 WRD =  PEEK (255): REM  PASS WORD BACK
TO APPLESOFT
 990 ER = 256 *  PEEK (254) +  PEEK (253):
REM GET SCORE
 1000  RETURN
 1010  REM ** TUTORIAL
 1020  HOME
```

```
 1030  VTAB 1: HTAB 3
 1040  INVERSE : PRINT "SPEECH RECOGNITION
DEMONSTRATION"
 1050  NORMAL : POKE 34,1
 1060  VTAB 6: HTAB 1
 1070  PRINT "THIS DEMONSTRATION IS MEANT TO
SHOW"
 1080  PRINT : PRINT "SOME ELEMENTARY WAYS TO
USE SPEECH"
 1090  PRINT : PRINT "RECOGNITION IN YOUR OWN
PROGRAMS."
 1100  VTAB 20: HTAB 1
 1110  PRINT "SAY 'OK' INTO THE MICROPHONE TO
CONTINUE"
 1120  GOSUB 940: REM  ML ROUTINE WAITS FOR A
SOUND HERE
 1130  HOME
 1140  VTAB 6: HTAB 1
 1150  PRINT "OBVIOUSLY, CONTROLLING PROGRAM
FLOW IS"
 1160  PRINT : PRINT "ONE SIMPLE WAY TO USE
IT.  INSTEAD OF"
 1170  PRINT : PRINT "HAVING TO REACH FOR THE
'RETURN' KEY TO"
 1180  PRINT : PRINT "CONTINUE, THE USER JUST
HAS TO SPEAK!"
 1190  GOSUB 1570
 1200  HOME
 1210  VTAB 1: HTAB 1: CALL  - 868
 1220  VTAB 1: HTAB 8: INVERSE
 1230  PRINT "TRAINING THE COMPUTER": NORMAL
 1240  VTAB 6: HTAB 1
 1250  PRINT "TO UNDERSTAND SPEECH THE COM-
PUTER MUST"
 1260  PRINT : PRINT "LEARN WHAT YOUR SPEECH
SOUNDS LIKE.": PRINT : PRINT
 1270  PRINT "CALLED 'TRAINING', WE DO THIS BY
SAYING": PRINT : PRINT "A WORD TO THE COMPUTER
MULTIPLE TIMES.": PRINT
 1280  GOSUB 1570
 1290  HOME : VTAB 6: HTAB 1
 1300  PRINT "YOU ARE ABOUT TO TRAIN YOUR
APPLE TO"
 1310  PRINT : PRINT "UNDERSTAND 7 DIFFERENT
WORDS.": PRINT
 1320  PRINT : PRINT "AS YOU SAY THE WORDS,
SPEAK NATURALLY"
 1330  PRINT : PRINT "AS YOU WOULD IN CONVER-
SATION.": PRINT
 1340  GOSUB 1570
 1350  HOME : VTAB 6: HTAB 1
 1360  PRINT "REMEMBER HOW YOU SAY THE WORDS
BECAUSE"
 1370  PRINT : PRINT "THE COMPUTER WILL TRY TO
IDENTIFY"
 1380  PRINT : PRINT "UNKNOWN WORDS BASED ON
THE WAY YOU"
 1390  PRINT : PRINT "PRONOUNCED THESE."
 1400  GOSUB 1570
 1410  TEXT : HOME : VTAB 1: HTAB 1
 1420  PRINT "PLEASE SAY THE FOLLOWING WORDS:"
 1430  PRINT
 1440  REM ** SYSTEM TRAINING HAPPENS HERE
 1450  FOR X = 0 TO MAX
 1460  PRINT "ONE": POKE WNUM,X: CALL RCRD
 1470  PRINT "TWO": POKE WNUM,X + MAX: CALL RCRD
 1480  PRINT "THREE": POKE WNUM,X + MAX * 2: CALL
RCRD
 1490  PRINT "FOUR": POKE WNUM,X + MAX * 3: CALL
RCRD
 1500  PRINT "FIVE": POKE WNUM,X + MAX * 4: CALL
RCRD
 1510  PRINT "SIX": POKE WNUM,X + MAX * 5: CALL
RCRD
 1520  PRINT "RETURN": POKE WNUM,X + MAX * 6:
CALL RCRD
 1530  PRINT
 1540  NEXT X
 1550  RETURN
 1560  REM ** SAY 'OK' PROMPT
 1570  VTAB 22: HTAB 9
 1580  PRINT "SAY 'OK' TO CONTINUE..."
 1590  GOSUB 940: RETURN
 1600  REM ** DO INIT
 1610  PRINT  CHR$ (4);"BLOAD EARS.OBJ"
 1620 WNUM = 255
 1630 MAX = 9: REM ** NUMBER OF COPIES OF EACH
WORD
 1640 RCRD = 24576
 1650 FIND = 24579
 1660 ELEVEL = 70
 1670 ARRAY$(0) = "1. DRAW BOX AROUND SCREEN"
 1680 ARRAY$(1) = "2. ERASE BOX AROUND SCREEN"
 1690 ARRAY$(2) = "3. CATALOG CURRENT DISK"
 1700 ARRAY$(3) = "4. BEEP SPEAKER"
 1710 ARRAY$(4) = "5. HIRES DEMO"
 1720 ARRAY$(5) = "6. QUIT"
 1730 QUIT$(0) = "1. NO, I DON'T WANT TO QUIT"
 1740 QUIT$(1) = "2. YES, I WANT TO QUIT NOW"
 1750  RETURN
```

Listing 2: EARS.S

```
    1 *********************************
    2 *                               *
    3 *            ][-Ears            *
    4 *      Speech Recognizer        *
    5 *                               *
    6 *           "EARS.S"            *
    7 *                               *
    8 *          c 1990 by            *
    9 *      By David Gauger II       *
```

```
10 *                              *
11 *      Merlin 8 Assembler      *
12 *                              *
13 ********************************
14
15          ORG   $6000
16
17 BIN1LO   EQU   $06 ;bins ptr lo byte
18 BIN1HI   EQU   $07 ;bins ptr hi byte
19 RAWLO    EQU   $08 ;raw data rcrd bffr
   pntr lo byte
20 RAWHI    EQU   $09 ;raw data rcrd bffr
   pntr hi byte
21 SLICE    EQU   $EB ;time slice # (0-3)
22 OUTBYTE  EQU   $EC ;temp storage of
   output byte
23 WORDLO   EQU   $ED ;current word's
   weighted diff,lo
24 WORDHI   EQU   $EE ;current word's
   weighted diff,hi
25 TEMPLO   EQU   $FA ;temp storage
26 TEMPHI   EQU   $FB ;temp storage
27 SCORELO  EQU   $FD ;lowest difference,
   lo
28 SCOREHI  EQU   $FE ;lowest difference,
   hi
29 WORDNUM  EQU   $FF ;# of word we're
   dealing with
30 BUTTON   EQU   $C061 ;button 1 hard
   ware loc
31
32 RECORD   JMP   RECORD1 ;record a word
33 FIND     JMP   FIND1   ;recognize word
34
35 * RECORD
36 * Enter with Wordnum holding $FF to
   record unknown word
37 * or a number from 0 to $FE showing
   which word to learn
38
39 RECORD1  JSR   RCRD     ;do recording
40          JSR   CONDENSE ;condense into
   32 bytes in table
41          RTS
42
43 RCRD     LDA   #$00
44          STA   RAWLO  ;init recbufl
45          LDA   #$62
46          STA   RAWHI  ;init recbufh
   one too low
47          LDY   #$00   ;init byte cntr
48 WAVEHI   BIT   BUTTON ;pos 1/2 of
   cycle?
49          BMI WAVEHI;yes wait until neg
50 WAVELO   BIT   BUTTON;neg 1/2 ofcycle?
51          BPL   WAVELO;yes wait for
   rising edge
52 PAGELUP  INC   RAWHI    ;next page
53          LDA   RAWHI    ;
54          CMP   #$67     ;done yet?
55          BEQ   RCRDDUN  ;yes
56 BYTELUP  LDA   #$01     ;
57          STA   OUTBYTE  ;init bit cntr
58 BITLUP   LDA   BUTTON   ;take a sample
59          ASL            ;hi bit=sample.
   Roll into carry
60          ROL   OUTBYTE;roll sample
   into outbyte
61          BCS   STORBYTE;must need to
   store the byte
62          LDX   #04  ;delay length
63 WAITLUP  DEX        ;do a small delay
64          BNE   WAITLUP ;
65          JMP   BITLUP ;and do it all
   again
66 STORBYTE LDA   OUTBYTE ;get outbyte
67          STA   (RAWLO),Y;store in input
   buffer
68          INY           ;are we done
   with page yet?
69          BEQ   PAGELUP  ;yes
70          JMP   BYTELUP  ;
71 RCRDDUN  RTS
72
73 * Condense - condenses raw data from 1K
   down to 32 bytes.
74
75 CONDENSE LDA   #$63     ;
76          STA   RAWHI    ;init input
   buffer ptr, hi
77          LDA   #$00     ;
78          STA   RAWLO    ;init input
   buffer ptr, lo
79          LDA   WORDNUM  ;get wordnum
   flag
80          CMP   #$FF    ;input unknown
   word?
81          BEQ   UNKNOWN  ;yes
82          JSR   INITWRD ;no-learn instead
83          JMP   LEARNIT  ;
84 UNKNOWN  LDA   #$62     ;put unknown
   word in right bffr
85          STA   BIN1HI   ;init bin
   pointer for input
86          LDA   #$E0     ;
87          STA   BIN1LO ;init bin ptr, lo
88          JSR   CLEAR  ;clear the area
89 LEARNIT  LDY   #$00   ;input counter
90          LDA   #$04 ;# of slices to use
91          STA   SLICE  ;init slice cntr
92          LDX   #$08   ;byte roll cntr
93          LDA   #$00   ;
94          STA   TEMPHI ;init result cntr
95
```

```
 96 * Positive half of wave
 97
 98 CLUPPOS  LDA   (RAWLO),Y ;get 8 samples
 99 CBITLUP  ASL            ;put a sample in
    carry
100         BCC   NEGBIT  ;must be neg
    half of wave now
101         JSR   NUCOUNT ;still positive:
    increment count
102         DEX           ;done w/ these
    samples yet?
103         BEQ   CBYTDUNP ;yes
104         JMP   CBITLUP  ;no - do again
105 CBYTDUNP LDX  #$08   ;re-init bit cntr
106         INY            ;point to next
    data position
107         BEQ   CPAGEDUN ;must be done
    with slice
108         JMP   CLUPPOS  ;do loop again
109
110 * Negative half of wave
111
112 NEGBIT   JSR   NUCOUNT  ;update result
113         DEX         ;done with these 8
    samples yet?
114         BEQ   CBYTDUNN ;yes
115         JMP   CBITLUPN ;no
116 CLUPNEG  LDA   (RAWLO),Y;get 8 more
117 CBITLUPN ASL           ;roll a sample
    into carry
118         BCS   WAVEDUN  ;must be done
    with the wave
119         JSR   NUCOUNT  ;not done yet:
    update result
120         DEX            ;done with
    these samples yet?
121         BEQ   CBYTDUNN ;yes
122         JMP   CBITLUPN ;no
123 CBYTDUNN LDX   #$08 ;re-init bit counter
124         INY        ;point to next data
    position
125         BEQ   CPAGEDUN ;must be done
    with page (slice)
126         JMP   CLUPNEG;do loop again
127 CPAGEDUN DEC   SLICE  ;are we done yet?
128         BEQ   ALLDUN;done condensing
129         INC   RAWHI ;No: point to next
    page of data
130         CLC      ;prepare for addition
131         LDA   BIN1LO ;get bin pointer
132         ADC   #$08   ;point to next
    set of bins
133         STA   BIN1LO ;put pointer back
134         LDY   #00    ;
135         LDX   #$08   ;init bit counter
136         JMP   CLUPPOS ;
137 WAVEDUN  LDA   TEMPHI  ;get wave's
lngth (#            of samples)
138         JSR   BINSORT ;sort into
    correctbin
139         LDA   #$00    ;
140         STA   TEMPHI;reinit reslt cntr
141         JMP   CBITLUP ;& do all again
142 NUCOUNT  INC   TEMPHI;update wavelength
    count
143         BEQ   OVER ;must have rolled
144         RTS
145 OVER     DEC   TEMPHI ;roll back to 255
146         RTS
147 ALLDUN   RTS
148
149 * (internal routine)
150 * Enter w/ A holding num to be sorted
151 * & Binllo/hi w/ base addr of thisbin
152
153 BINSORT  STY TEMPLO;store y temporarily
154         LDY   #$00   ;cntr: points to
    bins
155         CLC     ;prepare for compares
156         CMP   #4  ;Is it less than 4?
157         BCC   FOUNDBIN ;yes
158         INY      ;point to next bin
159         CMP   #5 ;is it 4?
160         BCC   FOUNDBIN ;yes
161         INY       ;point to next bin
162         CMP   #6  ;is it 5?
163         BCC   FOUNDBIN ;yes
164         INY       ;point to next bin
165         CMP   #8  ;is it 6 or 7?
166         BCC   FOUNDBIN ;yes
167         INY       ;point to next bin
168         CMP   #10 ;is it 8 or 9?
169         BCC   FOUNDBIN ;yes
170         INY       ;point to next bin
171         CMP   #12      ;is it 10 or 11
172         BCC   FOUNDBIN ;yes
173         INY        ;point to next bin
174         CMP   #20  ;between 10 & 19?
175         BCC   FOUNDBIN ;yes
176         INY            ;must be > 20
177 FOUNDBIN LDA   (BIN1LO),Y;get corr. bin
178         CLC ;get ready for addition
179         ADC   #1    ;inc bin's count
180         BEQ   BINROLL  ;> 255?
181         STA   (BIN1LO),Y ;restore bin
182         LDY   TEMPLO   ;restore y reg
183         RTS            ;
184 BINROLL  LDA   #$FF     ;max=255
185         STA   (BIN1LO),Y ;store as
    result
186         LDY   TEMPLO   ;restore y reg
187         RTS
188
189 * FIND
190 * Enter with data in recbuf, templates
    at locations in
```

```
191 * page 68, highest word to check
    against in WORDNUM.
192 * Result passed back in WORDNUM.
    Error (difference)
193 * passed back in SCORELO, SCOREHI
194
195 FIND1    LDA   #$FF     ;
196          STA   SCORELO  ;seed the
    score counter with
197          STA   SCOREHI ;the highest
    value possible.
198 FINDLUP  JSR   FINDIT  ;wordnum
    contains highest word
199          JSR   COMPARE ;16 bit
    compare to find smallest
200          DEC   WORDNUM ;pt to nxt word
201          LDA   WORDNUM ;get wordnum
202          CMP   #$FF    ;done yet?
203          BEQ   FOUNDIT ;yes
204          JMP   FINDLUP;no:try next word
205 FOUNDIT  LDA   OUTBYTE ;get result word
206          STA   WORDNUM ;pass back
    in wordnum
207          RTS
208
209 FINDIT   LDA   #$00    ;
210          STA   WORDLO  ;init word
    total, lo byte
211          STA   WORDHI  ;init word
    total, hi byte
212          STA   TEMPLO  ;init temp,
    lo byte
213          STA   TEMPHI;init temp, hi
    byte
214          LDA   #$62    ;
215          STA   RAWHI   ;init unknown
    word pntr hi
216          LDA   #$E0
217          STA   RAWLO   ;init unknown
    word pntr lo
218          JSR   WORDADRS ;set pointers
    to point in table
219          LDA   #$00    ;4 time slices
    (starting w/ 0)
220          STA   SLICE   ;init slice byte
    counter
221          LDY   #$FF    ;(count to 31
    bytes/word max.)
222          LDX   #$08    ;8 bytes of data
    per slice
223 FINDLUP1 INY          ;next byte
224          CPY   #32  ;done with entire
    word yet?
225          BEQ   BIGDUN  ;yes
226          LDA   (RAWLO),Y ;get a byte of
    data
227          CMP   (BIN1LO),Y ;which is
    bigger?
228          BCC   OTHER1  ;bin1lo is bigger
229          SEC           ;rawlo=bigger:
    prep to subtract
230          SBC   (BIN1LO),Y ;find diff
    (borrow never needed)
231 DUN1     JSR   ADD1 ;add diff to total
232          DEX  ;done w/ this slice yet?
233          BEQ   FINDDUN1  ;yes
234          JMP   FINDLUP1  ;no-do again
235 OTHER1   LDA   (BIN1LO),Y;get data byte
236          SEC    ;prepare for subtraction
237          SBC   (RAWLO),Y;find difference
238          JMP   DUN1  ;and do again
239 ADD1     CLC       ;prepare for addition
240          ADC   TEMPLO  ;add to total
241          STA   TEMPLO  ;save result
242          BCS   MORE1   ;we had a carry
243          RTS
244 MORE1    INC   TEMPHI ;account for carry
245          RTS
246 FINDDUN1 LDA   SLICE   ;get slice number
247          CMP   #$0     ;is it slice #0?
248          BEQ   NEWTTL  ;yes: give full
    weight in ttl
249          CMP   #$1     ;is it slice #1
250          BEQ   NEWTTL  ;yes: give full
    weight in ttl
251          CLC          ;no: divide by 2
    for 50% weight
252          ROR   TEMPHI   ;roll hi byte
253          ROR   TEMPLO   ;roll lo byte
    (carry rolls in)
254          CMP   #$2      ;is is slice #2
255          BEQ   NEWTTL   ;yes
256          CLC    ;no: divide again
    for 25% weight
257          ROR   TEMPHI   ;roll hi byte
258          ROR   TEMPLO   ;roll lo byte
    (carry rolls in)
259 NEWTTL   LDA   TEMPLO   ;get total
260          CLC          ;prepare for ad
261          ADC   WORDLO ;add to big total
262          STA   WORDLO ;save result, lo
    byte
263          LDA   TEMPHI ;get hi byte
264          ADC   WORDHI ;add hi bytes
    (carry adds)
265          STA   WORDHI;save result,byte
266          INC   SLICE  ;next slice
267          LDX   #$08   ;8 bytes per slice
268          LDA   #$00   ;
269          STA   TEMPLO ;re-init templo
270          STA   TEMPHI ;re-init temphi
271          JMP   FINDLUP1;& do all again!
272 BIGDUN   RTS      ;diff between words in
    wordlo,hi
273
274 COMPARE  LDA   WORDHI  ;get hi byte
```

```
275           CMP   SCOREHI ;lower than lowest
    so far?
276           BCC   CHANGIT ;yes: found a new
    lowest number
277           BEQ   MAYBE   ;if hi bytes =,
    check lo byte
278           JMP   COMPDUN  ;
279 MAYBE     LDA   WORDLO   ;get lo byte
280           CMP   SCORELO  ;are we lower?
281           BCC   CHANGIT  ;yes - save new
    lowest number
282           JMP   COMPDUN  ;no so leave
    everything alone
283 CHANGIT   LDA   WORDHI   ;get wordhi
284           STA   SCOREHI  ;save it
285           LDA   WORDLO   ;get wordlo
286           STA   SCORELO  ;save it
287           LDA   WORDNUM  ;get wordnum
288           STA   OUTBYTE  ;save it in
    outbyte for later
289 COMPDUN   RTS
290
291
292 * INITWRD (internal routine)
293 * Enter with WORDNUM holding number of the
    32 byte
294 * buffer to clear ($0-$FE).  Must clear
    out buffers
295 * before using the learn mode to store any
    data there.
296
297 INITWRD   JSR   WORDADRS ;set up pointer
298 CLEAR     LDY   #31      ;each buffer is 32
    bytes long
299           LDA   #00      ;
300 LUP1      STA   (BIN1LO),Y ;zero out data
    buffer
301           DEY            ;next location
302           BEQ   INITDUN  ;done yet?
303           JMP   LUP1     ;no
304 INITDUN   STA   (BIN1LO),Y ;account for y=0
    location
305           RTS
306
307 WORDADRS  LDA   #$67     ;
308           STA   BIN1HI;set pointer for hi
    bank
309           LDA   WORDNUM  ;get word number
310 ADRSLUP   CMP   #8       ;less than 8?
311           BCC   CALC     ;no
312           INC   BIN1HI   ;next page needed
313           SEC     ;prepare for subtraction
314           SBC   #$8      ;reduce by 8
315           JMP   ADRSLUP  ;
316 CALC      CLC          ;prepare for addition
317           ASL            ;multiply by 32
318           ASL            ;
319           ASL            ;
320           ASL            ;
321           ASL           ;
322           STA   BIN1LO   ;save in 0 page ptr
323           RTS
```

IIgs Animation

Illusions of Motion, Part III

by Stephen P. Lepisto
12907 N. Strathern
North Hollywood, CA 91605

Steve is a full time IIgs programmer who enjoys the rare status of freelancer. One of his multitudinous projects includes ***FirePower GS.***

In my last article, I presented a program that moved two different images over a complex background without disturbing each other or the background. In addition, the images had areas of transparency which allowed the background to show through them as they moved. I also introduced the notion of shadowing to provide flicker-free animation. This time, I will introduce that last basic component of computer animation: changing the image as it moves.

I'm sure there are times when you have played a game on the computer in which you guided a walking man or flaming ship through various perils. In the last two installments, I have presented the concepts needed for moving that man or ship across the screen. This month, I will show you how to make that man walk and that ship flame while they move (well, I won't actually show you the man or the ship but I will show you how to create a pulsating blue diamond, which is almost the same thing!).

Movement in Motion

Notice I use the phrase "walking man." In those sorts of games, a man doesn't really walk through the jungle (or castle or dungeon). Rather, a rectangular image with areas of transparency is being moved across a complex background that happens to resemble a jungle (or castle or dungeon). What makes the man appear to walk is the way the image changes as it moves. Instead of plotting the same image over and over (like we did in the last installment), a whole set of images are being plotted, one at a time, with each image of the set being just a little different from the others. Showing these subtly-changing images one after the other in rapid sequence gives the illusion of walking. Change the position where each image is plotted and you get a man walking across the screen. Just like in the movies.

So, to make a pulsating diamond, all we really need to do is plot different images instead of the same one, as we move the diamond around. However, for the illusion to work, we need to make sure the changes in the images are orderly and that they occur in a orderly and timely fashion. So, we need to introduce the concept of time to our little program. After all, if all the different images of the walking man were shown too quickly, the man would appear to be running in place! If the images occurred too slowly, the man would look like he was skating across the jungle. We need to pause for just the right amount of time between the showing of each image of the sequence to give the eye a chance to see that image on the background: this is why we need to introduce time to the program.

Programming Time

Time is relatively straightforward to add to a program. All it really amounts to is, a variable that changes at a regular rate. We then make sure the change in position and the change in the image occur in synchronization with that timing variable. For example, if we have a counter that is incremented once each time we execute the main loop of our program, and we cause the image being moved to be changed whenever the clock is incremented, the changing image would be in sync with the program. In prac-

tice, this approach is a little too simple because the image would change far too rapidly. Instead, we will change the image only after a certain number of clock ticks have passed. This will give us the proper delay in the changing image to see what is going on.

The main characteristic of a time counter is that it needs to be constant: it needs to change at regular intervals. A time counter can be implemented in a variety of ways, with the most common being use of interrupts or a simple counter incremented once through the main loop. What method is used will depend on what you are trying to achieve: te interrupt method is very accurate when you need to synchronize motion and change to a time-dependent event such as music or a real-time clock. The loop counter is useful when you want to synchronize everything in the program to everything else (since everything will be executed once each time through the main loop, it doesn't really matter how long it takes to do any one thing since everything else will wait for that one task to finish). In our example, we will be using a form of the loop counter approach.

Programming Change

The other aspect of changing an image in an orderly fashion is some method of accessing each image of a sequence in a given order. If you didn't get the order right, it might appear, for example, that the man was walking backward even though he was moving forwards. This is really bad for the eyes! So, we need a list of images and a variable that will act as an index into that list. That index will then be changed in sync with the timer.

The index that controls which image of the sequence is shown is often called a frame counter, since each image of a sequence is generally called a frame (taken from the movie industry where each image of the film is called a frame). This frame counter will be used to tell the program which frame to draw next.

How To Do It

For the timer in our program, we will be using a variant on the loop counter. Our timer will be a variable that is initialized to a specific value (called the Master Delay Value) and then decremented once each time through the main loop. When this counter reaches zero, we increment the frame counter which will cause the next image to be shown. We also reset the timer so it will count down to the next image change. Step by step, the process looks like this:

1) Initialize the time counter to the Master Delay Value.

2) Initialize the frame counter to 0 (start with first image in sequence).

3) Show the proper image in the sequence as referenced by the frame counter.

4) Decrement the timer by 1.

5) If the timer has reached 0, then

6) Reset the timer to the Master Delay Value.

7) Increment the frame counter by 1.

8) If the frame counter has gone past the last frame in the sequence, then reset the frame counter to 0.

9) Repeat steps 3 through 8.

Notice that we continue to display the image each time through the loop, even though the image itself isn't changed each time through the loop. This allows each image of the sequence to persist long enough for the eye to see it. This also allows us move the image each time the main loop is processed.

(As an aside here, note that it is possible to synchronize the motion to the timer in the same way the frame counter is synchronized if a slower motion is needed [in practice, moving once every time the main loop is executed is often too fast]. This is a different approach to velocity, in which we control speed by controlling when the change in position occurs as opposed to controlling the change in position itself.)

If we were to use an outside timer such as an interrupt from an actual clock (called Clock), then we would get rid of step 4 in the above algorithm and replace steps 1, 5, and 6 with something like this:

1) Initialize our timer to Clock plus Master Delay

Value.

5) If the Clock equals or exceeds our timer then

6) Reset our timer to Clock plus Master Delay Value.

The only change here is we usually don't want to reset the outside interrupt Clock to a Master Delay Value so we need to create a target value in our timer and wait for the interrupt Clock to catch up to that target value. When the Clock has reached or passed our target timer, we reset our counter to some future time.

Making It All Work

Okay, to bring it all together, we need a timer, a frame counter, and a sequence of images. This brings us to the changes to the program (which was built up over the last two installments):

1) Make the changes in listing 1. The lines to be added or changed are marked; the other lines are there to position the changes correctly. **Note that these changes assume you have the previous two installments (c.f. March and May, 1990).**

2) Add the code and data as shown in listing 2.

3) Replace the `draw_images` routine with the new one in listing 3.

4) Finally, add the new images in listing 4. Note that these new images replace the green square from the previous program.

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

Now, when you run this program, you should see two pulsating blue diamonds racing around the screen, one moving slower than the other.

In Conclusion

With this installment, you now have the basics of animation on the IIgs. You can move an image across a blank screen. You can move that image across a complex picture. You can even cause that image to change while it's moving. You can now create your own Illusions of Motion.

Things to Experiment With

1) In the default values (just after the init_images routine), the def_frametime array is the number of clock ticks between each frame of animation. Make this number smaller to have the diamond pulse more quickly; increase it for a slower pulse.

2) Look at the image sequence defined in image_0_list. Notice how the first two images are reused at the end of the sequence to make the pulsing look like its growing then shrinking (when the sequence is played over and over, the diamond grows and shrinks and grows in a regular rhythm). Play around with the order here to see the effects of that order. For example, delete or comment out the last two image references to see a different pulse effect.

3) To get a taste of some of the complexities that can arise with this type of animation, try to add a second image to this program. One of the headaches that often occurs in any program that uses complex animation is the organization of the image data and how to most efficiently access that data. Most of the changes needed to support a second sequence will take place in draw_images. Init_images will need to be changed as well. A hint: concentrate on those areas which reference image_0_list and figure out a way to reference multiple sequences.

4) I mentioned last time that there was a way to properly achieve velocities above two and that the changes needed occurred in only one routine. That routine is show_images. Basically, the limit of two on the velocity has to do with the two pixel border around the image. When the image is shadowed to the main screen, the border will erase the previous image at the same time as the new image is being shown (because that is the size of the rectangle being shadowed). If the velocity is greater than two, though, the shadowing rectangle is no longer large enough to completely erase the previous image.
If you were to enlarge the area being shadowed so it

took into account the velocity, the previous image would be erased no matter how far the new image was displaced by that velocity. However, this can slow things down as the shadowed area gets larger when the velocity gets larger. In addition, you have to watch out when an image gets close to the edge of the screen since you don't necessarily want to shadow memory that isn't visible.

This problem of shadowing moving images is one good reason why many programs chose to shadow the entire display area once after repositioning all the things that are moving and/or changing. Since shadowing such a large area takes no small amount of time, the size of the display area is made just small enough to minimize that delay in shadowing. This explains why many arcade-style games on the IIgs have such small play areas.

Listing 1:

In the following fragments, add the lines marked with a + at the end. Some of the routines have been truncated. This is indicated by ".....".

```
Animate
        .....

        jsr     draw_images
        jsr     show_images
        jsr     erase_images
        jsr     move_images
        jsr     update_counters              +
        lda     #1
        jsr     pause_a_moment
        jsr     read_key
        bcc     :1
        rts


init_images
        ldx     #0
:1
        stz     frame_count,x                +
        lda     #image_0_listx-image_0_list/4 +
        sta     frames_in_seq,x              +
        lda     def_frametime,x              +
        sta     frame_timer,x                +
        lda     def_velx,x
        sta     xvelocity,x;horizontal velocity
def_bytewidth   da 8,8
def_height      da 15,15
def_buffer      adrl buffer1,buffer2
def_frametime   da 6,6                       +


image_width     ds MAXIMAGES*2
image_bytewidth ds MAXIMAGES*2
buffer_adrs     ds MAXIMAGES*4
frame_count     ds MAXIMAGES*2               +
frames_in_seq   ds MAXIMAGES*2               +
frame_timer     ds MAXIMAGES*2               +
```

Listing 2:

Add this routine and data list to the code. I suggest you put the code just before the draw_images routine and the data list just before the images themselves.

```
* Update the time and frame counters for all
* sequences.

update_counters
        stz     image_index
:1      lda     image_index
        asl
        tax
        dec     frame_timer,x   ;decrement timer
        bne     :3

* When timer = 0, reset to Master Delay Value

        lda     def_frametime,x
        sta     frame_timer,x

*  and increment the frame counter.

        lda     frame_count,x

* When frame index reaches end of sequence,
* reset to beginning of sequence.

        cmp     frames_in_seq,x
        bcc     :2
        lda     #0
:2      sta     frame_count,x
:3      inc     image_index
        lda     image_index
        cmp     number_of_images
        bcc     :1
        rts


* Sequence list for the pulsating diamond.
```

```
image_0_list
        adrl    basic_image_0,basic_image_1
        adrl    basic_image_2
        adrl    basic_image_1,basic_image_0
image_0_listx

mask_0_list
        adrl    basic_mask_0,basic_mask_1
        adrl    basic_mask_2
        adrl    basic_mask_1,basic_mask_0
```

Listing 3:

Replace the draw_images routine with this new one. Since there were a number of changes to this routine, it is simpler to reenter it.

```
draw_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
        lda     buffer_adrs,y
        sta     buffer_ptr
        lda     buffer_adrs+2,y
        sta     buffer_ptr+2
        lda     frame_count,x
        asl
        asl
        tay
        lda     image_0_list,y
        sta     image_ptr
        lda     image_0_list+2,y
        sta     image_ptr+2
        lda     mask_0_list,y
        sta     mask_ptr
        lda     mask_0_list+2,y
        sta     mask_ptr+2
        jsr     buffer_image
        jsr     plot_image
        inc     image_index
        lda     image_index
        cmp     number_of_images
        bcc     :1
        rts
```

Listing 4:

Replace basic_image_1 and basic_mask_1 with the following additions.

```
basic_image_1
        hex     0000000000000000
        hex     0000000000000000
        hex     0000000000000000
        hex     0000000000000000
        hex     0000000440000000
        hex     0000004444000000
        hex     0000044444400000
        hex     0000444444440000
        hex     0000044444400000
        hex     0000004444000000
        hex     0000000440000000
        hex     0000000000000000
        hex     0000000000000000
        hex     0000000000000000
        hex     0000000000000000

basic_mask_1
        hex     0000000000000000
        hex     0000000000000000
        hex     0000000000000000
        hex     0000000000000000
        hex     0000000FF0000000
        hex     000000FFFF000000
        hex     00000FFFFFF00000
        hex     0000FFFFFFFF0000
        hex     00000FFFFFF00000
        hex     000000FFFF000000
        hex     0000000FF0000000
        hex     0000000000000000
        hex     0000000000000000
        hex     0000000000000000
        hex     0000000000000000

basic_image_2
        hex     0000000000000000
        hex     0000000000000000
        hex     0000000000000000
        hex     0000000000000000
        hex     0000000000000000
        hex     0000000000000000
        hex     0000000440000000
        hex     0000004444000000
        hex     0000000440000000
        hex     0000000000000000
```

```
        hex     0000000000000000
        hex     0000000000000000
        hex     0000000000000000
        hex     0000000000000000
        hex     0000000000000000

basic_mask_2
        hex     0000000000000000
        hex     0000000000000000
        hex     0000000000000000
        hex     0000000000000000
```

```
        hex     0000000000000000
        hex     0000000000000000
        hex     0000000FF0000000
        hex     000000FFFF000000
        hex     0000000FF0000000
        hex     0000000000000000
        hex     0000000000000000
        hex     0000000000000000
        hex     0000000000000000
        hex     0000000000000000
```

One of the Living Legends Returns... and GOSUBs, and...

MD-BASIC : A Review

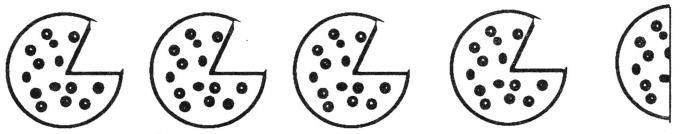
Reviewed by Jay Jennings

Editor: It's not that we don't ever want to run reviews, it's just that we only want to discuss extremely significant new heavy-duty hacker tools (we know our audience!). Applesoft itself is old technology, of course, but one of the reasons that MD-BASIC has so much potential is the principle of widest audience - Applesoft is built-in to every II. Publishers like Softdisk, for example, want to put as much on their disks as possible every month. Applesoft programs are good fodder for them because 16K of each one lives in ROM.

Jay Jennings is a full time Apple II freelance programmer (another of a rare breed). He is a former staff programmer for A2-Central and still does projects for them. For example, Jay is the seminar coordinator for the upcoming developer's conference in Kansas City.

The Jennings Pizza Rating for MD-BASIC:

4.5 (five pizzas possible)



What IS MD-BASIC, Anyway?

MD-BASIC can best be described as a cross-development system for Applesoft. Though MD-BASIC is a 16-bit compiler that runs on the Apple IIgs, it generates 8-bit Applesoft BASIC programs as its object files. The usual restrictions of Applesoft (two-letter variable names, line numbers instead of labels, and rudimentary looping structures) turn programming in BASIC into a giant headache. MD-BASIC changes that by eliminating the structure problems of Applesoft. With MD-BASIC, you write your programs with any text editor and then compile the source file into a runnable Applesoft program, a plain BAS file that you can RUN, LIST, or even compile further with the Beagle Compiler.

Sample MD-BASIC Program

Here's a short sample to give you a taste of the flavor of MD-BASIC code:

```
'_____________________________
' suck in some header defines from the
```

```
disk
'------------------------
#include <fileio.h>

'------------------------
' define some macros
'------------------------
#define EightyColumns print "^dPR#3" :
print
#define Locate(x,y)    htab x : vtab y

'------------------------
'program code starts here
'------------------------
EightyColumns       'go to 80 cols
home                'and clear the screen
Locate (35,12)
print "Hey, d00d!"

'------------------------
' now do a looping thing
'------------------------
x = 0          'initialize this variable
repeat
    x = x + 1
    print "^g"      'beep the speaker!!!
until x = 3

'------------------------
' and finally, some file I/O
'------------------------
gosub IO_Test
end

'------------------------
' the subroutine that does the file stuff
'------------------------
IO_Test:
     fOpen ("JayFile")
     fWrite ("JayFile")
          print "This is in my file!!!"
     fClose ("JayFile")
return
```

Compiling the above program gets you the following *(Editor: The control-D is invisibly embedded within the quote marks- a space saving feature)* :

```
 1  PRINT "PR#3": PRINT : HOME : HTAB 35:
VTAB 12: PRINT "Hey, d00d!":A = 0
 2  A = A + 1: PRINT "": IF  NOT (A = 3)
THEN 2
 3  GOSUB 4: END
 4  PRINT "open";"JayFile": PRINT
"write";"JayFile": PRINT "This is in my
file!!!": PRINT "close";"JayFile": RETURN
```

What It Does When It Does What It Does

Let's take a look at a few of the nifty source-level features of MD-BASIC. First, you can include comments in your programs in two ways: with the usual REM, or with an apostrophe. REM statements are included in the final compiled program (useful for copyright notices and so forth); comment statements that begin with an apostrophe are stripped out during compilation. You can use as many comment statements as you like without increasing the size of the resulting Applesoft file.

The first real line past the comment is a compiler directive that tells MD-BASIC to insert the file FILEIO.H into the source code. FILEIO.H is full of definitions that are used for, naturally enough, file I/O. It's essentially a set of macros that saves you from having to type PRINT CHR$(4) for each disk command. The fOpen, fWrite, and fClose calls used near the end of the sample program are defined in this file. (FILEIO.H, along with a number of other useful header files, is included with MD-BASIC.)

In the next section of our program, we define a couple of macros. (These are the kind of statements that FILEIO.H and the other header files contain.) We equate the symbol EightyColumns with the code segment PRINT "^DPR#3". We can then use EightyColumns as a command in our program, and MD-BASIC will convert it to PRINT "^DPR#3" during compilation. The caret (^) is used to denote a control character, just as you'd guess.

The REPEAT/UNTIL loop in the sample program could just as easily have been implemented with a FOR/NEXT loop, but there are some cases where REPEAT/UNTIL really comes in handy. MD-BASIC also includes WHILE/WEND, which differs from REPEAT/UNTIL in that the condition is tested at the top of the loop rather than at the end.

MD-BASIC doesn't use line numbers. Instead, you define labels for subroutines and call them by name.

You can name your file printing subroutine PrintFile and call it with GOSUB PrintFile, which is a lot easier on the gray matter than GOSUB 1027 or whatever. You don't have to worry about the line numbers changing when you add and delete lines, either.

Further Goodies

MD-BASIC also includes a decompiler, which allows you to create an MD-BASIC source file from an existing Applesoft file. Tweaking the resulting source file to take advantage of MD-BASIC's features is a good idea, but even simply decompiling and recompiling a program produces a smaller Applesoft file that runs faster. This is because the Applesoft code that MD-BASIC produces is highly optimized. A program I wrote about three years ago is 7987 bytes long as I originally wrote it; after simply decompiling and recompiling, the program was 6952 bytes long. MD-BASIC saved me over 1K, all without any work to speak of.

Early versions of MD-BASIC ran under the APW and ORCA/M shells, but the current version can be run standalone as well. A confusing, complex, and very powerful text editor (microEmacs) is included, but you can use whatever editor you like to edit your programs. Also included is AmperWorks, an ampersand package that gives Applesoft some much-needed capabilities such as copying files, listing text files, and searching and sorting arrays. The 30 commands in AmperWorks are worth the price of MD-BASIC all by themselves; with MD-BASIC it's an incredible deal and adds considerable power to your programming. (MD-BASIC includes header files to integrate AmperWorks seamlessly into the language.)

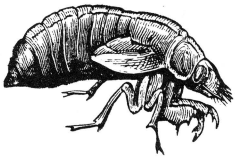
The documentation is a clear and concise 120-page bound manual that includes sections on the compiler, decompiler, advanced features, and AmperWorks. I'd ask for more source code examples, but I'd ask the same of any development system no matter how many examples were included. An included installation program will install the compiler, decompiler, and header files as a standalone application or as utilities in the APW and ORCA/M shells.

MD-BASIC has many other great features, but they're too numerous to detail in this review. I suggest you get ahold of the MD-BASIC demo and put it through its paces (it'll be on this month's 8/16 disk). You can also get it from most online services, and from Morgan Davis' BBS, Pro-Sol, at (619) 680-5379.

To sum it all up, MD-BASIC revives tired old Applesoft and makes it once again a viable development language for the Apple II. It's a great value at $49.95.

Morgan Davis Group
10079 Nuerto Lane
Rancho San Diego, CA 92078-1736
(619) 670-0563

Insecticide



We discovered yet another piece of code stuck underneath our chair that should have been included in Matt Neuberg's Sweet 16 article in our May issue. As many of you noticed, the relocation routine referenced in the article and JSR'd to by the main program was conspicous by its absence. Here 'tis, with proper line numbers:

```
354 *--------------
355
356 * OMIT what follows if you don't want pg 3
reloc
357
358           ORG
359
360 RELOCATE LDX   RELEND-SW16
361 RELONE    LDA   $900-1,X ;where SW16 is b4
reloc
362           STA   $300-1,X ;where we want it to
go
363           DEX
364           BNE   RELONE
365           RTS     ;Sweet16 is now ready for
use
```

SouthPaw : Changing Your Orientation

by Jason Blochowiak

Jason is heavily into INITs, having written BreakCursor (turns your cursor to an "X" when your machine crashes), and AnimatedWatch (which moves the hands of the watch cursor). He is currently working on the "Great American Software Project" and will probably retire soon thereafter.

SouthPaw is a short permanent initialization file (PIF) that changes the orientation of the arrow mouse cursor to something more natural for left-handed people. I'm not left-handed, nor is the person who suggested the idea to me (Evan Ron Aussenberg), but it seemed like it'd be an interesting hack. The program is a good introduction to the rather obscure art of tool-vector patching, as well as changing the cursor. Additionally, the source code is presented in both APW and MPW IIgs formats, so that you may compare the two.

The program is broken up into one code segment, main, and one data segment, Cursors. main has two parts: the first part tool patch installation code, which gets executed when SouthPaw is loaded at boot time and installs the tool vector monitor. This code (the second part of main) watches all tool calls made and gets control of the system when _InitCursor is called.

The Installation

One makes tool calls on the Apple IIgs by loading the X register with the tool call number, and performing a JSL to either $E1/0000 or $E1/0004. Those vectors point to a dispatcher within the tool locator, which finds the appropriate code and transfers control to it. Because of Apple's prudent decision to hold the entry point to the tool locator in a vector, it's possible to change what code is executed when a tool call is made.

Actually making the vector point to our code is relatively simple. (See the label InstallVectors.) First, the previous contents of each vector is saved. Next, we copy a jump to our code into the vector (at the label InstallOurs). This is done for each of the two vectors. Note that interrupts are temporarily disabled by the php/sei/plp instructions in order to prevent code running during the interrupt to call a half-formed address. (This would happen if an interrupt occurred after the sta >ToolVec, but before the sta >ToolVec+2. This is highly unlikely, but Murphy's law dictates that we play it very safe when dealing with interrupts.)

The Tool Vector Monitor

Upon entry to the primary vector monitor (the code that gets called when $E1/0000 gets called) at MyToolEnt, the carry flag is cleared, and the code drops into ToolEnt. When the secondary vector monitor gets called (when $E1/0004 is called) at MyToolEnt1, the carry flag is set, and the code branches to ToolEnt. At ToolEnt, the P register is pushed on the stack, so it can be determined later (from the carry flag) which of the saved vectors to use.

"Note that the method of tail-patching used here works only if the call we're patching has no parameters on the stack."

The X register is then compared against the one tool call number that we care about, _InitCursor ($CA04). If it doesn't match, control goes to VecNorm, which restores the P register. At VecNorm, if the carry flag is set, control goes to the location

where the old value of the alternate vector is stored; conversely, if carry is clear, the code drops into the old value of the primary vector. Either way, the code that was pointed to before we did our patch gets executed and our routine is finished.

If X does hold $CA04, indicating that an _InitCursor call was made, we pop P off of the stack and JSL to VecCall, which checks the carry flag and calls the code pointed to by the values held previously by the tool vectors. The result of this is that the original _InitCursor code is executed, and then SouthPaw regains control. (This technique, of getting control of a tool call after it has executed, is know as back-patching or tail-patching. Note that the method of tail-patching used here works only if the call we're patching has no parameters on the stack—if _InitCursor went looking for parameters, they'd be in the wrong place, and the call would fail miserably.)

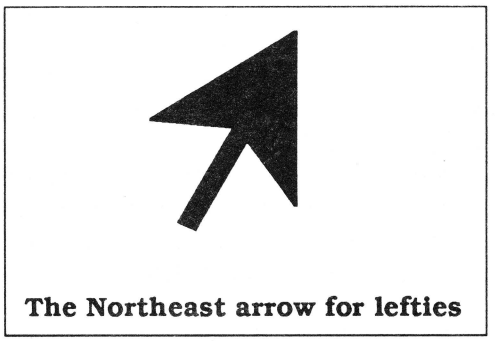
When the old code returns to us, we do our business (detailed below in "The Hack"), and then execute an RTL. Because another tool call is made right before returning, the carry flag will be set appropriately—in this case, always clear. (In other cases, we'd have to set the flag ourselves.)

You may have been taken aback at making another tool call within the tool vector monitor, since we've patched the tool dispatcher vector! This is called, simply enough, re-entrance. Code that can handle being re-entered is called (surprise) re-entrant. The monitor must be re-entrant; if it weren't, tool calls made during an interrupt would cause the system to fail, as might tool calls made from within another tool.

One simple way of avoiding problems with re-entrant code is by using the stack instead of address-based variables. For example, instead of setting the carry flag and then saving it on the stack to remember if the primary or alternate tool vector got called, a value could be stored in a variable. However, if an interrupt came in after storing the value and one of the interrupt routines made a tool call, it would cause a different value to be stored in that variable. The interrupted tool vector monitor would then call the incorrect tool locator code, which would be a Bad Thing.

The Hack

After the monitor determines that _InitCursor has been called, and after allowing the normal code to be executed, it's time to do the work. First, a call is made to _GetMasterSCB and the result is pulled off the stack into the accumulator. Then X and Y are loaded with a pointer to the 640 mode cursor. Next, the accumulator is ANDed with mode640, to determine if QuickDraw is in 640 mode. If it is, the code branches to Is640; if it isn't, X and Y are loaded with a pointer to the 320 mode cursor.



The Northeast arrow for lefties

Now, in either case, the X and Y registers hold a pointer to the new cursor for the current screen mode, so they're pushed on the stack as input for _SetCursor. _SetCursor is called and the cursor is set the new image.

The Translation

SouthPaw was originally written using the APW assembler on the IIgs. Since writing it, I purchased a Macintosh IIci to make writing programs for the IIgs easier. I now use the Macintosh Programmer's Workshop (MPW) IIgs cross-development tools as my development platform. If you're serious about programming the IIgs, and you can afford it (which is not a minor consideration), the MPW IIgs tools are something that you should look into. For this article, I thought that it would be interesting to present the source in both formats, so I converted the APW source to MPW IIgs source and it appears in listing two.

The conversion was done in four stages. First, Apple File Exchange was used to get the source file into the

Mac's file format (HFS). AsmCvtIIgs, an MPW tool that came with the assembler, was run to get the source into something that AsmIIgs would even start to look at. Some work was done with the editor and the shell to convert all those nasty spaces into tabs, and to make a few other minor cosmetic adjustments. Then I did a teeny bit of programming to make my life easier - I added a segment at the beginning which does a brl to branch around the data segment. This allows me to escape from the drudgery of declaring forward references. (More on this below.)

The specific differences between the files follow: one is named SP.S. I used ".S" as a suffix with APW. The Macintosh file is named sp.aii, since ".aii" is an enforced suffix with the MPW IIgs tools.

The keep directive is missing from sp.aii, as it's unnecessary. Additionally, instead of using mcopy with a pre-generated macro file, I directly "include" M16.QuickDraw and M16.Util into sp.aii. I also use "include" for the tool equates in sp.aii, but in SP.S, I use "copy" to import the tool equates.

The equates defined at the start of each file are similar. Though APW uses gequ and MPW IIgs uses equ, in this case, both assemblers treat them as the same.

You may notice that all of the code in main is the same. This isn't always the case, as there are some differences in syntax between the two environments. For example, MPW IIgs uses "val << bits" for a left shift and "val >> bits" for a right shift in expressions, while APW uses "val | bits" to shift left and "val | -bits" to shift right.

The data inside Cursors is the same, but the declaration is a bit different. In APW, you use the opcode field for the operation, and the operand field for the data format and size (for example, dc i2'3,5' states "define a couple of constant integers that are two bytes each [word-sized]: 3 and 5"). In MPW IIgs, you use the opcode field for the operation and the size of the individual operands (the code above would translate to dc.w 3,5, saying "define a couple of constant words, 3 and 5"). The MPW IIgs constant declarations dc.b x (define byte x), dc.w x (define word x), and dc.l x (define longword x) correspond to APW's dc i1'x' (define a constant that's one byte long), dc i2'x' (define a two byte constant [word-sized]), and dc i4'x' (define a four byte constant [longword-sized]), respectively.

Now, for a moment, let's look back at the routine at the label "skip". The MPW IIgs assembler requires that you declare all forward references. Rather than bother with declaring all of the references, I declare main, which I consistently use as an entry point, and then create a code segment which merely branches to main. This lets me put all of the code after all of the data, eliminating the need to individually declare any forward data references.

Although the MPW IIgs environment is significantly more powerful than APW, it's not always faster. I timed a full rebuild using both APW and MPW IIgs.

On my TransWarped IIgs with two megabytes of memory, I used the APW assembler and ZapLink with all files on :RAM5, running under System Software 5.0.2, it took 7 seconds for the first and the second full builds (with one immediately following the other).

I used v1.0 of the MPW IIgs assembler, and the version of LinkIIgs that came with v1.1b of the MPW IIgs tools on my Mac (with five megabytes of memory). The development system was on one SyQuest 45Mb removable hard drive, and the source and object code were on a second SyQuest drive. I used System Software 6.0.4 and MultiFinder 6.1b was active, but only the Finder was running in the background. It took 25 seconds for MPW IIgs to build the entire program the first time, and 11 seconds the second time.

There are a few important factors to consider while looking at these times. The first of these is the fact that the APW version used a macro file generated with MacGen, whereas the MPW version just included the appropriate macro files, wholesale. Additionally, APW got to do its job on a RAM disk. Finally, and probably most importantly, the MPW IIgs tools don't take much longer on larger files. As an extreme example of the difference in speed, Scott Lindsey, one of the programmers at Claris who has worked on AppleWorks GS, claims that doing a full rebuild of AWgs takes a number of hours on a Mac II, but it took a couple of days (days!) to rebuild when the AWgs team was still using APW.

The Rambling

As should be obvious, you could easily change the cursors that I used to something else—whatever you want. When creating the cursors, I usually create the one for 320 mode first because it's easier, due to the fact that each pixel takes up one hexadecimal digit. I then convert it to 640 mode by turning each $00 in the image into a $0, each $0f into a $3, each $f0 into a $c, and each $ff into a $f. (Note that this method won't work if you're working with a color cursor.)

You could also fix a small problem with SouthPaw: if a program were to do a _SetCursor with a cursor that was identical to the cursor set by _InitCursor, SouthPaw wouldn't be able to tell and the user would see the right-handed cursor. To get around this, you could intercept _SetCursor, and compare the image to the standard arrow cursor. If it matched, you could replace the address on the stack with a pointer to the appropriate left-handed cursor.

Listing one: APW source code

```
;
;       SP.S
;       "SouthPaw"  v1.0
;       an InitCursor() hack to make arrow
;       better for lefties
;
;       Copyright (c) 1990 by Jason Blochowiak
;       and Ariel Publishing. Some rights
;       reserved.
;
;       ** APW Assembler **
;


        keep  SP              ;OBJ file = SP
        case  on              ;case sensitive
        mcopy SP.Mac          ;get macro file

        copy  2/AInclude/E16.QuickDraw ;& equs

; Tool calls we have to watch for
InitCursor gequ $ca04

; Tool Vectors to intercept
ToolVec  gequ  $e10000
ToolVec1 gequ  $e10004

main    start                 ;start of segment
        using Cursors         ;with this as data

InstallVectors anop
        php
        phb
        phk
        plb                   ;make sure that
        long                  ;  interrupts
        sei                   ;  don't kill us!

        lda   >ToolVec        ;get main
        sta   VecSave         ;  tool vector
        lda   >ToolVec+2
        sta   VecSave+2

        lda   >ToolVec1       ;get alternate
        sta   VecSave1        ;  tool vector
        lda   >ToolVec1+2
        sta   VecSave1+2

InstallOurs anop
        lda   MyTool1         ;install my alt.
        sta   >ToolVec1       ;  tool vector
        lda   MyTool1+2
        sta   >ToolVec1+2

        lda   MyTool          ;install my main
        sta   >ToolVec        ;  tool vector
        lda   MyTool+2
        sta   >ToolVec+2

        plb
        plp
        lda   #0
        clc
        rtl

MyTool  jmp   >MyToolEnt      ;copied to
MyTool1 jmp   >MyToolEnt1     ;  tool vectors

MyToolEnt1 anop
        sec
        bra   ToolEnt
MyToolEnt anop
        clc
ToolEnt anop
        php
        cpx   #InitCursor     ;is this our call?
        bne   VecNorm         ;  no...

        plp                   ;  yes!
        jsl   VecCall

        PushWord #0           ;space
        _GetMasterSCB
        pla

        ldx   #Arrow640
        ldy   #^Arrow640
```

```
         and   #mode640
         bne   Is640

         ldx   #Arrow320
         ldy   #^Arrow320

Is640    anop
         phy
         phx
         _SetCursor
         rtl

VecNorm  anop
         plp
VecCall  anop
         bcs   VecSave1
VecSave  ds    4
VecSave1 ds    4

DoIt     anop
         bra   VecNorm

         end                    ;end of main

Cursors  data                ;start of data seg.

Arrow320 dc    i'11,4'
         dc    h'000000000000 0000'
         dc    h'0000000000f0 0000'
         dc    h'000000000ff0 0000'
         dc    h'00000000fff0 0000'
         dc    h'0000000ffff0 0000'
         dc    h'000000fffff0 0000'
         dc    h'00000ffffff0 0000'
         dc    h'0000fffffff0 0000'
         dc    h'000000ff0ff0 0000'
         dc    h'00000ff00000 0000'
         dc    h'000000000000 0000'

         dc    h'0000000000ff 0000'
         dc    h'000000000fff 0000'
         dc    h'00000000ffff 0000'
         dc    h'0000000fffff 0000'
         dc    h'000000ffffff 0000'
         dc    h'00000fffffff 0000'
         dc    h'0000ffffffff 0000'
         dc    h'000fffffffff 0000'
         dc    h'0000ffffffff 0000'
         dc    h'0000ffff0fff 0000'
         dc    h'0000fff00000 0000'

         dc    i'1,10'

Arrow640 dc    i'11,3'
         dc    h'00000000 0000'
         dc    h'00000c00 0000'
         dc    h'00003c00 0000'
         dc    h'0000fc00 0000'
         dc    h'0003fc00 0000'
         dc    h'000ffc00 0000'
         dc    h'003ffc00 0000'
         dc    h'00fffc00 0000'
         dc    h'000f3c00 0000'
         dc    h'003c0000 0000'
         dc    h'00000000 0000'

         dc    h'00000f00 0000'
         dc    h'00003f00 0000'
         dc    h'0000ff00 0000'
         dc    h'0003ff00 0000'
         dc    h'000fff00 0000'
         dc    h'003fff00 0000'
         dc    h'00ffff00 0000'
         dc    h'03ffff00 0000'
         dc    h'00ffff00 0000'
         dc    h'00ff3f00 0000'
         dc    h'00fc0000 0000'

         dc    i'1,10'

         end                    ;end of Cursors
```

Listing two: MPW IIgs source code

```
;
;        sp.aii
;        "SouthPaw"  v1.0
;        an InitCursor() hack to make arrow
;        better for lefties
;
;        Copyright (c) 1990 by Jason Blochowiak
;        and Ariel Publishing. Some rights
;        reserved.
;
;        ** MPW IIgs Assembler **
;

    case    on          ;case sensitive

; Macros
    include 'M16.QuickDraw' ;grab macro files
    include 'M16.Util'

; Equates
    include 'E16.QuickDraw' ;and equates file

; Tool calls we have to watch for
InitCursor equ $ca04

; Tool Vectors to intercept
```

```
ToolVec  equ  $e10000
ToolVec1 equ  $e10004

    entry   main:CODE;a forward CODE reference

skip    proc            ;start of procedure

    brl main

    endp                ;end of procedure
Cursors record          ;start of record (data)

Arrow320
    dc.w    11,4
    dc.b    $00,$00,$00,$00,$00,$00,$00,$00
    dc.b    $00,$00,$00,$00,$00,$f0,$00,$00
    dc.b    $00,$00,$00,$00,$0f,$f0,$00,$00
    dc.b    $00,$00,$00,$00,$ff,$f0,$00,$00
    dc.b    $00,$00,$00,$0f,$ff,$f0,$00,$00
    dc.b    $00,$00,$00,$ff,$ff,$f0,$00,$00
    dc.b    $00,$00,$0f,$ff,$ff,$f0,$00,$00
    dc.b    $00,$00,$ff,$ff,$ff,$f0,$00,$00
    dc.b    $00,$00,$00,$ff,$0f,$f0,$00,$00
    dc.b    $00,$00,$0f,$f0,$00,$00,$00,$00
    dc.b    $00,$00,$00,$00,$00,$00,$00,$00

    dc.b    $00,$00,$00,$00,$00,$ff,$00,$00
    dc.b    $00,$00,$00,$00,$0f,$ff,$00,$00
    dc.b    $00,$00,$00,$00,$ff,$ff,$00,$00
    dc.b    $00,$00,$00,$0f,$ff,$ff,$00,$00
    dc.b    $00,$00,$00,$ff,$ff,$ff,$00,$00
    dc.b    $00,$00,$0f,$ff,$ff,$ff,$00,$00
    dc.b    $00,$00,$ff,$ff,$ff,$ff,$00,$00
    dc.b    $00,$0f,$ff,$ff,$ff,$ff,$00,$00
    dc.b    $00,$00,$ff,$ff,$ff,$ff,$00,$00
    dc.b    $00,$00,$ff,$ff,$0f,$ff,$00,$00
    dc.b    $00,$00,$ff,$f0,$00,$00,$00,$00

    dc.w    1,10

Arrow640
    dc.w    11,3
    dc.b    $00,$00,$00,$00,$00,$00
    dc.b    $00,$00,$0c,$00,$00,$00
    dc.b    $00,$00,$3c,$00,$00,$00
    dc.b    $00,$00,$fc,$00,$00,$00
    dc.b    $00,$03,$fc,$00,$00,$00
    dc.b    $00,$0f,$fc,$00,$00,$00
    dc.b    $00,$3f,$fc,$00,$00,$00
    dc.b    $00,$ff,$fc,$00,$00,$00
    dc.b    $00,$0f,$3c,$00,$00,$00
    dc.b    $00,$3c,$00,$00,$00,$00
    dc.b    $00,$00,$00,$00,$00,$00

    dc.b    $00,$00,$0f,$00,$00,$00
    dc.b    $00,$00,$3f,$00,$00,$00
    dc.b    $00,$00,$ff,$00,$00,$00
    dc.b    $00,$03,$ff,$00,$00,$00
    dc.b    $00,$0f,$ff,$00,$00,$00
    dc.b    $00,$3f,$ff,$00,$00,$00
    dc.b    $00,$ff,$ff,$00,$00,$00
    dc.b    $03,$ff,$ff,$00,$00,$00
    dc.b    $00,$ff,$ff,$00,$00,$00
    dc.b    $00,$ff,$3f,$00,$00,$00
    dc.b    $00,$fc,$00,$00,$00,$00

    dc.w    1,10
    endr                ;end of record

main    proc            ;start of procedure
    with    Cursors     ;use Cursors as data

InstallVectors
    php
    phb
    phk
    plb                 ;make sure that
    long                ;  interrupts
    sei                 ;  don't kill us!

    lda >ToolVec        ;get tool vector
    sta VecSave
    lda >ToolVec+2
    sta VecSave+2

    lda >ToolVec1       ;get alt tool vector
    sta VecSave1
    lda >ToolVec1+2
    sta VecSave1+2

InstallOurs
    lda MyTool          ;install my tool vector
    sta >ToolVec
    lda MyTool+2
    sta >ToolVec+2

    lda MyTool1         ;install my alternate
    sta >ToolVec1       ;  tool vector
    lda MyTool1+2
    sta >ToolVec1+2

    plb
    plp
    lda #0
    clc
    rtl

MyTool
    jmp >MyToolEnt      ;copied to tool vectors
MyTool1
    jmp >MyToolEnt1

MyToolEnt1
    sec
```

```
    bra ToolEnt
MyToolEnt
    clc
ToolEnt
    php
    cpx #InitCursor      ;is it the call we
    bne VecNorm          ;  care about? no...

    plp                  ;  yes!
    jsl VecCall
    PushWord #0          ;space for result
    _GetMasterSCB
    pla

    ldx #Arrow640
    ldy #^Arrow640

    and #mode640
    bne Is640

    ldx #Arrow320
    ldy #^Arrow320
Is640
    phy
    phx
    _SetCursor
    rtl

VecNorm
    plp
VecCall
    bcs VecSave1
VecSave
    ds.l    1
VecSave1
    ds.l    1

DoIt
    bra   VecNorm

    endp                 ;end of procedure

    end                  ;end of source code
```

GENESYS

NEW!

GS Sauce RAM Card

Now available and shipping!

Genesys™...the premier resource creation, editing, and source code generation tool for the Apple II GS.

Genesys is the first Apple IIGS CASE tool of its kind with an open-ended architecture, allowing for support of new resource types as Apple Computer releases them by simply copying additional Genesys Editors to a folder. Experienced programmers will appreciate the ability to create their own style of Genesys Editors, useful for private resource creation and maintenance. And Genesys generates fully commented source code for ANY language supporting System 5.0. Using the Genesys Source Code Generation Langugage (SCGL), the Genesys user can tailor the source code generated to their individual tastes, and also have the ability to generate source code for new languages, existing or not.

Genesys allows creation and editing of resources using a WYSIWYG environment. Easily create and edit windows, dialogs, menu bars, menus menu items, strings of all types, all the new system 5.0 controls, icons, cursors, alerts, and much more without typing, compiling, or linking one single line of code.

The items created with Genesys can be saved as a resource fork or turned into source code for just about any language. Genesys even allows you to edit an existing program that makes use of resources.

Genesys is guaranteed to cut weeks, even months, off program development and maintenance. Since the interface is attached to the program, additions and modifications take an instant effect.

Budding programmers will appreciate the ability to generate source code in a variety of different languages, gaining an insight into resources and programming in general. Non-programmers can use Genesys to tailor programs that make use of resources. Renaming menus and menu items, adding keyboard equivalents to menus and controls, changing the shape and color of windows and controls, and more. The possibilities are almost limitless!

Genesys is an indispensable tool for the programmer and non-programmer alike!

Retail Price: $150.00

SSSi is pleased to announce that we will be carrying the GS Sauce memory card by Harris Laboratories. This card offers several unique features to Apple //gs owners:

- Made in USA
- Limited Lifetime Warranty
- 100% DMA compatable
- 100% GS/OS 5.0 and ProDOS 8 & 16 compatable
- Installs in less than 15 seconds!
- Low-power CMOS chips
- Uses "snap-in" SIMMs modules - the same ones used on the Macintosh
- Recycle your Macintosh SIMMs modules with GS Sauce.
- Expandable from 256K to 4 Meg of extra DRAM

This card is 100% compatable with all GS software and GS operating systems. It is 100% tested before shipping and has a lifetime warranty. The CMOS technology means that it consumes less power and produces less heat thus making it easier on your //gs power supply. There are no jumpers, just simple to use switches to set the memory configuration. One step installation takes less than 15 seconds.

Memory configurations:

Apple //gs model	add these:	total GS RAM
256K (ROM 1)	(1) 256K SIMM	512K
	(2) 256K SIMMs	768K
	(4) 256K SIMMs	1.25 Meg
	(1) 1 Meg SIMM	1.25 Meg
	(2) 1 Meg SIMMs	2.25 Meg
	(4) 1 Meg SIMMs	4.25 Meg
1 Meg (ROM 3)	(1) 256K SIMM	1.25 Meg
	(2) 256K SIMMs	1.50 Meg
	(4) 256K SIMMs	1.78 Meg
	(1) 1 Meg SIMM	2.0 Meg
	(2) 1 Meg SIMMs	3.0 Meg
	(4) 1 Meg SIMMs	5.0 Meg

Please note that you can not mix 256K and 1 Meg SIMMs packages on the same GS Sauce card, and that expansion must be performed in (1), (2) or (4) SIMMs modules.

Pricing:

We are offering a limited time "get acquainted" offer to our customers. The GS Sauce card is available from SSSi as:

0K	$89.95 - use your own 256K or 1 Meg SIMMs modules
1 Meg	$179.95
2 Meg	$269.85
4 Meg	$449.75

☞ **We are making a special offer to our Genesys users:**

Buy Genesys and and get a coupon to purchase GS Sauce for:

0K	$79.95 - use your own 256K or 1 Meg SIMMs modules
1 Meg	$159.90
2 Meg	$239.85
4 Meg	$399.75

We hope you will see what an excellant value the GS Sauce card is: low power consumption, SIMMs technology, inexpensive, made in USA and lifetime warranty!

Call or write for seperate 256K and 1 Meg SIMMs modules to upgrade your GS

Order by phone or by mail. Check, money order, MasterCard, Visa and American Express accepted. *Please add $5.00 for S/H*

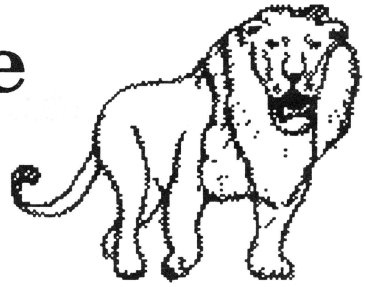
Simple Software Systems International, Inc.
4612 North Landing Dr.
Marietta, GA 30066
(404) 928-4388

MasterCard VISA AMERICAN EXPRESS

SSSi

PushPin Studios, St. Augustine, FL

KAT will sell no drive before it's time...



KAT will not ship a hard drive without first:

- Conferring with you about your entire system and setting the drive's interleave so as to insure optimal preformance *for you.*
- Discussing the various partioning options and then *setting them up to fit your specifications.*
- Depositing 20 megabytes of freeware, shareware, the latest system software, and all sorts of bonus goodies on the drive.
- Testing the drive for 24 hours before shipping it out.

KAT drives come in industrial-quality cases that have 60 watt power supplies (115-230 volts), cooling fans, two 50 pin connectors and room for another half-height drive or tape back-up unit. We also include a 6 ft. SCSI cable to attach to your SCSI card. You get all of this plus a one-year warranty on parts and labor!

SB 48 Seagate 48 meg 40ms	$549.99
SB 85 Seagate 85 meg 28ms	$698.99
SB 105 Quantum 105 meg 12 ms	$849.99

Looking for an even *hotter* system? Call and ask for a quote on our 170, 300, & 600 megabyte Quantum drives!

So ya wanna build yer own? Let KAT provide you with the finest parts available...

SB Case 2 HH Drives 7w 5h 16d	$139.99	**T-60 Tape** Teac 60 meg SCSI	$449.99
ZF Case 1 HH Drive 10w 3h 12d	$169.99	with hard drive	$424.99
48 meg HD Seagate 40 ms 3.5" SCSI	$349.99	**3.5" to 5.25" Frame**	$ 12.50
85 meg HD Seagate 28 ms 5.25" SCSI	$469.99	**Cable** 25 pin to 50 pin 6 ft.	$ 19.99
105 meg HD Quantum 12 ms 3.5" SCSI	$669.99	50 pin to 50 pin 6 ft.	$ 19.99

Programmers! Check our prices on your favorite development packages and accessories...

Byte Works

Orca C	$89.99
Orca M	$44.99
Orca Pascal	$89.99
Orca Disassembler	$34.99

Other software and accessories:

Vitesse, Inc.
Excorciser, virus detection system $ 29.95
Renaissance, hard disk optimizer $ 34.95
Guardian, program selector and disk utilities $ 34.95

Applied Eng. Transwarp GS	$289.99
Keytronic 105 Key ADB Keybrd	$139.99

Roger Wagner Publishing

Hyperstudio	$94.99
Macromate	$37.99

Stone Edge Technologies

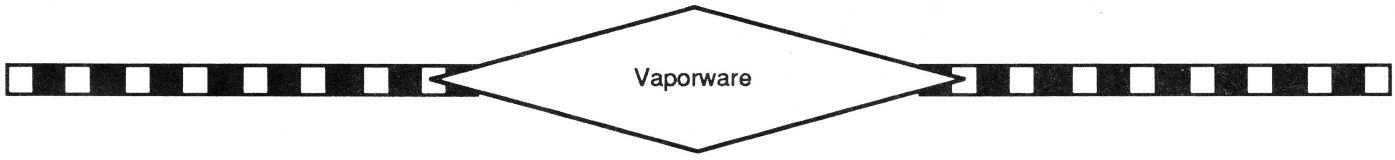
DB Master Pro	$219.99

Quickie, terrific hand scanner (400 dpi, 16 grays) $249.99

Computer Peripherals
ViVa24, 2400 baud, 100% Hayes compatible modem (comes with a FIVE YEAR Warranty) $139.99

1 meg SIMMs 80 ns	$89.99
1 meg X 1 80 ns	8/$79.99

Call the KAT at (913) 642-4611 or write: KAT, 8423 W 89th St, Overland Park, KS 66212-3039



by Murphy Sewall

From the June 1990 APPLE PULP,H.U.G.E. Apple Club (E. Hartford) News Letter
P.O. Box 18027,East Hartford, CT 06118

Apple IIgs - Mac Merger?
Apple is beta testing a number of products intended to, eventually, make the distinction between the Macintosh and the older II line functionally irrelevant. IIgs Operating System 6 and IIgs HyperCard (yes, they really are in beta test) will make the IIgs appear more Mac-like than ever. A "Multi-Finder IIgs" which allows hard disk space to be used as virtual RAM also is nearing marketable shape. Future Apple computers are likely to offer compatibility with older software by using the technology of the //e on a chip which manages the I/O and video on the Mac IIfx (Did everyone catch John Sculley's reference to the "Macintosh IIgs" during Apple Vision 90's for educators on April 24)? Apple has, but has not committed to market, a Mac Plus board for the IIgs as well as the 16 MHz 68000 "under $2,000" color (under $1,500 black and white) Mac SE compatible with Apple II coprocessor. An Apple II board for the Mac II family also is in beta test. Apple executives may still be trying to decide which options to offer and when to announce them. Sources indicate they can only continue to hum a tune which sounds vaguely like "September Song." - found in my electronic mailbox

Not PCjr.
In August, IBM will once again attempt to penetrate the home market. This time Big Blue will endeavor to avoid a repeat of the PCjr failure by offering practical power at street prices as low as $1,000. MS-DOS, Microsoft Works, and Prodigy software will be bundled with the 10 MHz 80286-based AT-bus "Bluegrass" desktop (see last July's column) from their typewriter division. The list price will be between $1,300 and $2,000 depending on configuration. The display will support VGA; a 1.44 Mbyte 3.5 inch drive and 640K of RAM are standard. There are no expansion slots, but options do include a built-in 2400 baud modem, a 30 Mbyte hard disk, and a mouse. - PC Week 23 April

Lap Size LapMac.
Apple and Toshiba are working on a four to six pound Macintosh laptop to replace the current overpriced Mac Anvil, er Portable. - InfoWorld 23 April

Zip's IIgs Accelerator.
Zip Technology is beta testing a 12 MHz accelerator for the Apple IIgs which contains only 22 chips (compared to more than 200 on the older, slower Applied Engineering accelerator). Alas, the problem is that although the hardware zips, marketing doesn't. It may be some time before Zip ships. - found in my electronic mailbox

New NeXT.
Steve Jobs says that NeXT will offer a Motorola 68040 based workstation with a very high resolution color monitor and at least six new applications by Christmas. Among the workstation's features will be built-in modem and fax capabilities. Mr. Jobs also said that owners of the present 68030 model will be able to upgrade motherboards for $1,495. Motorola claims the 25 MHz 68040 at 20 MIPS is 10 time faster than the 68030 and outperforms the 25 MHz Intel 486. - InfoWorld and PC Week 14 May

Laser GS.
Video Technologies is telling dealers in Canada that their 10 MHz Apple IIgs clone (see last September's column) which was shown to developers last July will be for sale by Christmas. - found in my electronic mailbox

Multiple Emulations.
A reader of last month's column's touting of A/UX's ability to run UNIX, Macintosh, and MS-DOS software noted that

the the new Amiga 3000 (see last February's column) will be able to run UNIX, AmigaOS, and MS-DOS. A Macintosh board is available for the Amiga but requires Mac ROM chips which are not easy to come by. Apparently, UNIX for the Amiga has been delayed until fall.

PostScript for the Masses.
Adobe Systems plans a major rewrite of its page description language and a family of inexpensive PostScript controllers. Pat Marriott, the firm's director of marketing, says "within 18 months, Adobe's goal is to offer OEMs a range of controllers to bring users PostScript printers for under $1,000." Meanwhile, Apple plans to hold increase the performance of its LaserWriter line by offering
a faster version of the IINTX within a few weeks. - PC Week 7 and 14 May

Mac System 7.0 Delayed Until Year's End
Roger Heinen, Apple's vice president of software development, has reversed earlier assertions that the new Macintosh operating system announced last year is "on schedule" (see last month's column). Developers were told last month that "by New Year's, almost all of our users should have an opportunity to upgrade to System 7.0." Tony Meadows, former director of a Northern California Mac
developers group translated "by New Year's" as meaning next January's MacWorld. A key feature of System 7 which Apple wants all developers to use is a set of application standards referred to as Interapplication Communication (IAC). IAC is designed for seamless communication among applications which should make it easier to build hybrid applications which collaborate with one another. - InfoWorld 14 May

New HyperCard.
HyperCard 2.0, a major rewrite, will be announced on June 26 and will ship with all Macs starting in July. Version 2.0 features variable card sizes, multiple windows, and a "style text" feature compatible with True Type, better printing capabilities, and enhanced HyperTalk. Version 2.1 will ship with System 7.0 and will offer Apple Events scripting and Mac Apps via HyperTalk. - InfoWorld 14 May

1-2-3 For Windows.
Lotus has announced an intention to deliver a 1-2-3 product for Microsoft Windows 3.0 which will offer the core spreadsheet functions of 1-2-3 version 3.0 along with the look and feel of the 1-2-3/G Presentation Manager version. In the interim, Lotus plans to ship version 3.1 during the third quarter. The update will incorporate PC Publishing's Impress program to provide WYSIWYG graphics publishing and drawing features. - PC Week 7 and 14 May

Flash in the Pan.
Don't expect any upgrades of bug fixes for Flash, Beagle's only Macintosh product. The only Mac programmer on Beagle's staff has left to work on Photo Shop for Adobe Systems (said to be an improved version of Pixel Paint Professional). Flash's future is limited anyway because Macintosh System 7.0 will offer it's major functions. - found in my electronic mailbox

Upgrades One of These Days.
dBase IV version 1.1, the alleged "bug fix," will appear this month and will still have bugs in it. Xyquest, which had announced an upgrade of Xywrite for the first quarter 1990 (did you miss it?), will delay shipping until late summer or early fall in order to add more features. Software Publishing plans to unveil two new DOS versions and an OS/2 Presentation Manager version of its popular Harvard Graphics by the end of the year. Harvard Graphics 2.3, an update to version 2.12, is expected in late June. Version 3.0 is slated for release in the fourth quarter along with the OS/2 PM version. Word Perfect Corporation officials have confirmed that a version of their popular word processor is forthcoming for the Windows 3.0 environment. While there is no definite shipment date, managers said they expect to deliver the Windows version within six months of the release of the forthcoming OS/2 Presentation Manager product. Word Perfect also is working on a scaled-down version of Word Perfect 5.1 called Letter Perfect for laptop users and others who don't need all the features of version 5.1 - InfoWorld 23 and 30 April and PC Week 7 May

From the House of Ariel

• *8/16 on Disk* •

The magazine you are now holding in your hands is but a subset of the material on the *8/16* disk. We have combed the BBS's and data services across the country to collect the best of the public domain and shareware offerings for programmers. Not only that, but we have extra articles and source code written by our staff. With DLT16 and DLT8 (**D**isplay **L**auncher **T**hingamajigs) to guide you, you can read articles, display graphics, and even launch applications. **NOTE: DLT16 requires GS/OS v 5.02 on your system.**

Highlights (so far every disk has had more than 650K of material!):

- **March '90:** 8 bit - the entire source code to Floyd Zink's Binary Library Utility. 16 bit - Bill Tudor's fantastic InitMaster CDEV, Parik Rao's Orca/APW utilities
- **April '90:** 8 bit - SoftWorks, an AppleWorks™ filecard interface for Applesoft programs, the source code to Bruce Mah's File Attribute Zapper. 16 bit - More Orca and APW utilities, Phil Doto's APF viewer
- **May '90:** 8 bit - Tom Hoover's AppleWorks Style Line Input. 16 bit - Bryan Pietrzak's shell utilities for Orca/APW, Steve Lepisto's "Illusions of Motion".
- **June '90:** 8 bit - 3D graphics package, MicroDot™ Demo, DiskWorks, 80 column screen editor. 16 bit - Assembly Source Code Converter (shareware), Install DA (on the fly; by our our own Eric Mueller), Find File source code.

1 year - $69.95 6 months - $39.95 3 months - $21 Individual disks are $8.00 each

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

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

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

• *ProTools*™ •

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

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

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

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

ProTools is $39.95 (your choice of 3.5" or 5.25" disks).

• *Back issues of The Sourceror's Apprentice* •

Ross's Recommendations:

8 bit:

Feb '89 - Relocation Without Dislocation, by Karl Bunker
...techniques for writing relocatable 8 bit code
Jan, Mar, Apr, Aug '89 - The Applesoft Connection Parts 1-4, by Jerry Kindall
...using the ampersand vector and internal Applesoft routines. A classic series.
Jun '89 - Peeking at Auxiliary Memory: A Monitor Utility, by Matthew Neuberg
...lets the monitor display aux mem, an invaluable 128K programming tool.
Sep '89 - Getting More Value(s) From Your Game Port, Eric Soldan
...increase range of values returned by a joystick for DHR coordinates, etc.

16 bit:

Jan '89 - Programming with Class 1, by Jay Jennings
...an introduction to GS/OS class 1 calls
Mar & Jun '89 - Vectored Joystick Programming, by Stephen Lepisto
...a technique for increasing responsiveness in reading the joystick
July '89 - Making a List (and checking it twice), by Ross W. Lambert
...an introduction to the GS List Manager
Sep '89 - Generic Start II, The Sequel, by Jay Jennings
...an introduction to the new start up song and dance for new system software
Jan '90 - Trapping Tricky Tool Errors, by Jay Jennings
...a classy programmer's error trap for the GS.

All back issues are $3.00 each (postage and handling included except for non-North American orders. Those of you on other shores please add $1.50 extra per issue).

Our guarantee: Ariel Publishing guarantees your satisfaction with our entire product line (software *and* publications). If you are *ever* dissatisfied with one of our products, we will cheerfully refund the amount you paid on your request. To order, just write to: **Ariel Publishing, Box 398, Pateros, WA 98846 or call (509) 923-2249.**

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

MasterCard VISA

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

Your satisfaction is our guarantee!

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