June, 1990
Volume 1, No. 4

8/16

The Journal of Apple II Programming $3.50

Magical Resources: Joe Jaworski tackles the Resource Manager

In this issue:

Computer	Title	Author	Page
both	The Publisher's Pen	Ross Lambert	3
re:	*DB Master Professional Programming, KCFest, Welcome Cecil*		
IIgs	Magical Resources	Joe Jaworski	7
re:	*Using the IIgs Resource Manager*		
both	A Modest Proposal	Ross Lambert	22
re:	*Using the Pascal Protocol for parameter passing*		
both	Vaporware	Murphy Sewall	30
re:	*News, Views, and Ruminations industry-wide*		
8 bit	Taking a Screen Test	Jerry Kindall	31
re:	*Saving text screens to disk*		
IIgs	Directing Traffic	Nate Trost	37
re:	*Using tdc, tcd, etc. to swap direct pages*		

Nite Owl's

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

Battery Replacement Kit for Apple IIGS Computer

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

Patent Pending

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

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

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

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

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

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

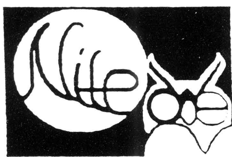
We highly recommend that every IIGS owner keep a spare battery on hand, ready for when the inevitable battery failure occurs. These Lithium batteries have a shelf life of over 10 years. The Slide-On kits come with a full 90 day satisfaction guarantee.

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

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

Photo-Copyable

Quantity • Pricing	
1+	14.95
10+	12.00
50+	10.00
100+	9.00
Sales Tax	
Kansas residents 6%	
Shipping•Handling	
Add $2.00 / Order	
Overseas add $5.00	



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

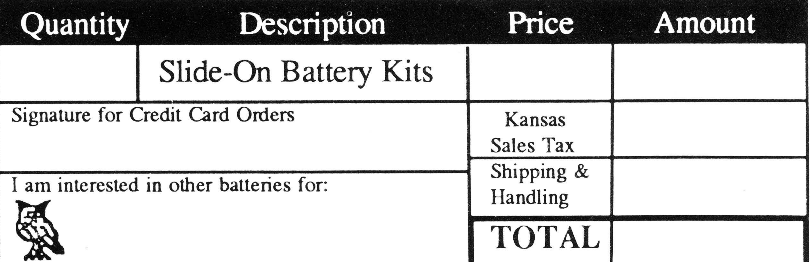
(Cut & Paste Address Label)

Ship to:

Telephone #:

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

Credit Card or PO# | Expiration Date



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

Prices may Change without notice.

8/16

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

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

Introductory subscription prices in US dollars:

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

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

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

WARRANTY and LIMITATION of LIABILITY

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

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

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

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

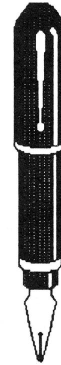
AppleWorks is a registered trademark of Claris, Corp.

ZBasic is a registered trademark of Zedcor, Inc.

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

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

The Publisher's Pen



by Ross W. Lambert

It has been my intention to make this column a short and punchy exposition of non-technical information for highly technical people. This month is neither short nor punchy, but every item is important (well, most of them, anyway). And if you'll excuse my unbridled optimism, there is so much to report because so much is happening in the Apple II world.

Welcome Aboard, Cecil!

• Let's have an official *8/16* welcome to Cecil Fretwell, long time *CALL A.P.P.L.E.* contributor, editor, spokesperson, and all around "character" (I was going to call him the granddaddy of the Apple II, but I figured he'd skewer me with a fork the next time he saw me...). Cecil is joining our list of regular contributors. You'll notice that he is not appearing in print this month, however - that's because he is reassuming the role of answer man, and is now dutifully awaiting your letters and questions. Yes, Mike Rochip has passed the torch to a new generation (now the question remains: who is older, Mike or Cecil?). Send your questions to us here (Box 398, Pateros, WA 98846); we'll forward them on to Cecil. Just make sure and mark them ATTENTION: GRANDDAD.

Hehehe.

• Speaking of Cecil, he and Ken Kashmarek have recently done translations of the source code in Ron Lichty and David Eye's *Programming the Apple IIgs* into C and Merlin 16+ assembly, respectively. You can get either or both versions directly from Ron Lichty (or buy the book outright, if you haven't yet): *Ron Lichty, P.O. Box 27262, San Francisco, CA 94127.* Note that the C disk is $20, the Merlin 16+ disk is $10, and the book is $32. Also note that the C disk contains C source for *Hello, World* for almost every point at which the book suggests that source can be assembled and linked, from

Chapter 5 - 9. The Merlin disk, on the other hand, only contains the completed *Hello, World* source. However, Ken created two versions. In one he went line by line and converted the tool calls to Merlin supermacros, thereby greatly compacting the source. In the other he left the calls expanded. The information I received from Ron Lichty also mentioned that he will send out a free errata page for the book correcting two bugs, and error in the text, and one serious typo - but you need to be sure to send him a self-addressed, stamped envelope.

The Genesis of GeneSys (& DesignMaster)

• IIgs programming took a quantum leap forward last month with the release of both DesignMaster from the ByteWorks and GeneSys from SSSi. I plan on formally reviewing both products as soon as possible, but in a nutshell they allow you to "draw" your interface by moving around windows, buttons, etc. as objects on the screen. They are hot, hip, and happening. One overlooked aspect of these programmer productivity tools is that they are also good learning tools. I do not have a copy of DesignMaster yet so I cannot comment about its language support, but GeneSys spits out code for Micol Advanced BASIC GS , Merlin 16+, and a veritable plethora of other languages and environments.

So What Software also has a nice set of editors in their Call Box package that they've been shipping for a long, long time. These small, task specific editors are handy (though they don't compare in overall power and integration with GeneSys). Unbeknownst to many, the Call Box editors can generate APW output, too, and Bill Stevens tells me that they are adding Merlin 16+ output soon. The Call Box package is definitely a useful alternative if you primarily feel at home in Applesoft. Bill has some interesting surprises up his sleeve for the future, too.

Based on preliminary information, GeneSys will have a suggested retail price of around $150, but SSSi was reported to be selling it on special at AppleFest for approximately $100. I don't know how long that price lasts. DesignMaster will retail for $95, but the Byte Works is introducing it for $55. The entire Call Box package (which includes the Call Box Toolbox Programming System), retails for $99.

The Byte Works
4700 Irving Blvd NW Suite 207
Albuquerque, New Mexico 87114

SSSi
4612 North Landing Drive
Marietta, GA 30066
(404) 928-4388

So What Software
10221 Slater Ave., Suite 103
Fountain Valley CA 92708
(714) 964-4298

• A couple folks have had questions about the timing of the magazine and the disk. At present, we are sending them out **separately.** The magazine is sent to the printer for reproduction and binding in the middle of the month prior to distribution. At that time we begin assembling the files for the disk (and hopefully grab a few moments to upgrade the two DLT packages). When we get the magazine back from the printer (7-10 days later), we drop everything and ship it out to y'all third class .

Mr. Postman tells me that domestic third class should take 10-14 days, meaning that most of you should be getting your magazines around the end of the first week of each month. Let me know if it is taking longer. We could push back production to better accomodate you. Anyway, after the magazine goes out, we finish the disk and send them to you first class. In theory they should arrive sometime close to the magazine since they are going on a faster boat.

In theory.

• The producer's of GS Numerics (now distributed by

A2-Central, though I wonder how it fits into their product mix) have a marketing lesson for all of us who sell our own 'wares: They are marketing bulldogs - they don't ever give up or let go. I've received a press release, a free promo video, and two follow up phone calls.

I don't fully endorse the software or their marketing methods (a video?), but there is a point to learn here. If we labor long into the night making our program perfect, how can we expect to sell any of them if we don't work just as hard at getting the word out? The GS Numerics folks have learned that lesson and generated a LOT of free publicity thereby. You and I both *can* do likewise.

Inquiring Minds Want to Know

• In spite of popular demand, we are printing a one page rumor column, distilled from the industry-wide ramblings of one Murphy Sewall. Murph's column is syndicated in a rather odd, typically '90s sort of way. It is published in the Washington Apple Pi and on various online networks. We have picked it up and reprinted it (with the author's permission, of course) because it is interesting and much more serious (i.e. based on knowledge) than other columns of similar ilk.

Besides, we're all rumor mongers at heart. I think it'll be fun. And it might even tip you off to a trend or market movement that will make you money (there's a method to our madness, eh?).

Last one to Kansas City is a rotten egg!

Marketing geeks (of which Tom Weishaar and I are guilty of being from time to time) are often guilty of hyperbole. Not so with Tom's ad for the Kansas City Developer's Conference in late July. I'd never tell *him* this, but I'd have plastered a promo for it for free since it is so vitally important to Apple II developers (I made him pay for an ad, instead). The Apple II Developers Association was formed there last summer and the Beagle Bros. also bared the soul of the TimeOut series.

This year promises to be even better. My friend and cohort Jay Jennings is arranging the conference seminars, so I'm privy to some of the classes in the works (as if the IIgs College were not enough!). You want state of the art IIgs sound and animation? You want to find out how to push 8 bit Apple's so hard that their monitor's break out in a sweat? You want to find out how to market your software (or, better yet, get other people to market it?).

It's all there. And more. Be there. Aloha.

Wait! I feel a pulse!

Okay, so now that I'm a member of the established press nobody tells me anything "secret" anymore. But I look and I listen. The Apple II is about to be revived. It was never really dead, only neglected and abused. One fact I know: Apple is about to launch the first Apple II retail ad campaign in years. And my sources tell me that the passionate letters of Apple II fans everywhere are reaching the right people up at the big house. Keep those cards and letters comin'.

"If we labor long into the night making our program perfect, how can we expect to sell any of them if we don't work just as hard at getting the word out?"

Kersey, Bert..... 392-7645

Uh, I'm not **literally** looking for Bert Kersey. Y'all can quit sending me his phone number and address. It was all a multi-issue dip into metaphorical humor (with a point). Just humor me. In spite of my abuse of the

English language (my apologies to my colleagues), I were an English teacher. Some people tell jokes. I indulge in metaphor.

DB Master Professional Programming

In case you couldn't tell, I've got a lot on my mind this month. One such burden is the fantastic DB Master Professional package produced by Stone Edge Technologies. You may wonder what that relational database has to do with Apple II programming.

Two very important things.

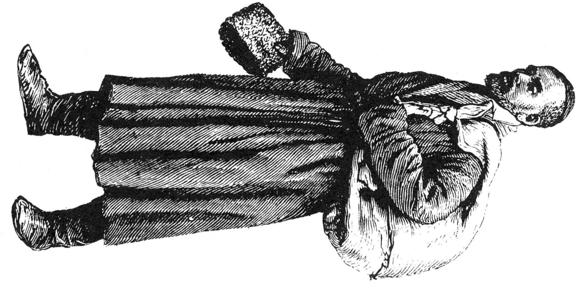
First, in the MS-DOS and Macintosh worlds there is an entire breed of developers known as database applications programmers. Contrary to what you might think, these guys are not writing databases. They are *using* databases like Dbase, Foxbase, and 4th Dimension. These muscular relational packages allow you to create a stand alone, task specific or turnkey application that these database applications programmers then sell for a fortune (but they pay a hefty license, of course).

A friend of mine routinely rakes in $2000 for a week long "consultation" in which he creates a database for a specific (usually small) business. He basically just creates templates of their paperwork and then generates reports that manipulate the data upside down and backwards.

It is a kind of programming, only it is based on a highly specific knowledgebase, not technical wizardry.

Don't get me wrong, I'm all for technical wizardry. But if the Apple II is ever going to gain a foothold in the small business market (which it should, especially if a faster GS is in the works), we need people to create these kinds of specific applied applications to show the MS-DOS bigots how powerful the II really is. Subpoint 1: every relational database I've ever messed with is pretty much disk-based. This makes disk access speed one of the most important factors in determining the true end-user perceived speed of the product. Any Apple II with a good high speed hard disk is a competitive machine when you cast things in these terms. Subpoint 2: Far too many people assume that if AppleWorks™ can't do it, it's time to look into MS-DOS. I think these folks would be quite surprised at the speed and flexibility of DB Master Professional.

I don't bless every facet of the program. The inconsistency of the interface bothers me. But creator Barney Stone has not claimed to have the last word on such matters, thus he created the subject of main point #2:



The BASIC Programmer's Pak

Barney has distilled the essence of his file management system into a set of Applesoft ampersand routines which allow you to read and write records to an existing DB Master Professional data file. This opens the door for us to create a background system with the main application, and then stand alone data entry templates in Applesoft.

But that's not all folks. Barney is also creating a complete stand-alone applications generator. Like Dbase, this would allow you to setup data entry templates and reports within the program and then compile it into a regular SYS file.

I think this is an exciting new opportunity for developers and even hobbyists with an eye towards turning a buck. And there's one HUGE, market sitting around aching to be reached: schools. If we hesitate, this opportunity will pass us by because "...the only decent administrative software is on the IBM/Macintosh" (said by my local high school principal). But a good attendance/business office/student records kind of program could be written in two or three days using DB Master Professional, and then marketed to **thousands** (maybe even millions) of schools who have not yet jumped to other brands.

I think this is sufficiently important to run a series of articles illuminating the details. Let me know what you think about that (and our overall editorial mix). Though I don't promise to answer or print every note, I really do read them all.

== Ross ==

Magical, Mysterious Resources

by Joe Jaworski

Probably the most talked about (and misunderstood) component of System Disk 5.0.2 is the resource fork. After all, these little devils have really given us some grief: not only do we have to learn another hundred or so tool calls, but they have rendered those old copy programs we love into useless applications. Now we're supposed to change the way we program entirely, developing windows parameters, controls, and even simple pascal strings as separate entities from our main program. We need new support software and tools in addition to our compilers, and what's really scary is the thought anybody with a resource editor will be able to modify our programs extensively without having the source code. Is all this really worth it?

Yes, it is. The Resource Manager is the greatest thing since sliced bread. It unloads a tremendous amount of programming burden to the system software, making our programs load faster, run faster, and manage memory better. Best of all, it does all of this with less programming effort on our part.

Sounds too good to be true? Stick with me. We'll cover the basics, move to some advanced topics, then show a resource-based application in action. Hopefully, the mysteries of the resource will fall by the wayside.

Resources are too simple

A resource or extended file is nothing more than two files in one. I'll repeat that one more time because this is most important: a resource or extended file is nothing more than two files in one. These two files live behind a single filename. Like any other ProDOS file, the file can live in any folder or subdirectory. We'll call one half of the extended file FILE1 and the other FILE2.

When you access an extended file using a GS/OS READ or WRITE command, you normally get access to FILE1. If you want work with FILE2, you have to specifically ask for it (via setting a bit in the parameter table). Then, everything works the same. You can read or write anything you want to either FILE1 or FILE2. For example, you could save a graphics image in FILE1 and ASCII text in FILE2.

FILE1 and FILE2 are truly independent files, just like two files on the disk. Writing to one doesn't disturb the data in the other, and vice versa.

FILE1 and FILE2 can also be different sizes. FILE1 could be 512 bytes in size, while FILE2 could be 128,000 bytes. When you CATALOG a disk, you see the total size of FILE1 and FILE2 listed in the blocks column of the catalog display. It doesn't matter if you use AppleSoft or the Finder to do the catalog; the two halves of the file are "added up" and you see the real size of the file in blocks. However, the end-of-file or file length field of the catalog display only shows you the length of FILE1 (this is to maintain compatibility with ProDOS 8).

Speaking of ProDOS 8, this "dual file personality" of extended files has presented some conflicts with older programs, especially disk utilities. As you may know, all ProDOS files have a "storage type" which defines the size and structure of the file. A new storage type was created for the extended file. Older copy programs probably won't recognize this new storage type and will generate an error. Some that don't behave well under error conditions try to proceed anyway, and usually copy only FILE1. Sometimes, they completely miss the mark and only copy garbage. The point is, using an older ProDOS 8 program to copy extended files is not a good idea.

"Using an older ProDOS 8 program to copy extended files is not a good idea."

You can determine which copy programs you have are

compatible with extended files and which are not. To be safe, do the following experiment on a floppy and not on your hard disk. Try and copy an extended file, such as the System file SYS.RESOURCES located in the System.Setup folder. If you get an "Unsupported storage type" error, you know there's a compatibility problem. If no errors occur during the copy, compare the block count of the original and duplicate file. If they're different, the program doesn't support extended files.

ProDOS 8 MLI commands don't directly support extended files. However, if a ProDOS 8 program interprets directory information itself, it can be made "extended file compatible". There is nothing inherent in ProDOS 8 that prevents an application (i.e., SYS file) from working with extended files. Any program, be it 8 or 16 bit-based, can work with extended files.

Metamorphosis: Extended Files Become Resources

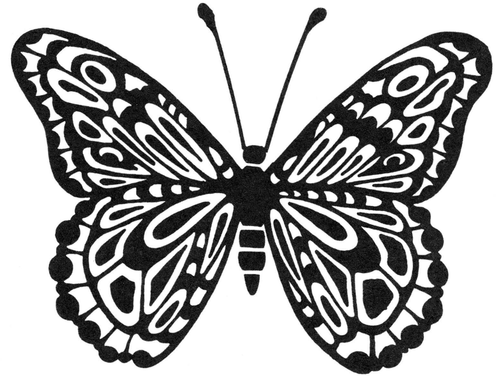
Up until now, I've purposely avoided the word "resource" in describing the extended file. This is because we've only discussed extended files, and NOT resource files.

Here's the trick: to convert an extended file into a resource file, you simply pass a filename to the Resource Manager's _CreateResourceFile call. That's all there is to it. The Resource Manager changes the original storage type to an extended type, creates a blank FILE2, and then writes some special data to FILE2. (Through all of this, the Resource Manager hasn't touched FILE1 and does not modify it's data.) With one tool call, you have just transformed a conventional file into an extended file as well as converted it into a resource file.

The special data written to FILE2 by the Resource Manager is nothing more than a list of pointers and fields that describe how other data will be eventually stored within FILE2. In a sense, this data is a "mini catalog" that points to and defines all the resources held within FILE2.

Note that none of the Resource Manager's tool calls can access or otherwise disturb the data in FILE1. (That half of the file normally holds the executable code of your application.)

It's probably become apparent that the Resource Manager toolset works more like GS/OS file calls rather than conventional toolbox calls. This is true. In fact, the Resource Manager has an OPEN call for files, and uses its own form of file reference numbers similar to that of GS/OS and ProDOS 8 MLI commands.



Now let's change some of our terms to adhere with the rest of the world. We will call FILE1 the data fork, and FILE2 the resource fork. This is the standard conventions used to describe an extended file, either created or modified by the Resource Manager. By the same token, we will call our "extended" file a "resource" file. The two terms are often used interchangeably, though technically a resource file is an extended file with the resource fork formatted or initialized by the Resource Manager.

A resource itself is a chunk of data that is stored in the file's resource fork. The Resource Manager will automatically grow and shrink the size of the resource fork to hold more or less resources. Each individual resource can vary in length, depending upon what kind of data you want to store in it.

For most applications, you really don't have to be concerned with the "mini catalog" or special pointers contained at the beginning of the resource fork. The Resource Manager manipulates this section exclusively, just as GS/OS manipulates subdirectories when you ask for a file.

If you want to access the data within a particular resource, you simply pass a type and ID to the Resource Manager. These two numbers define a unique resource within the fork, just as a full pathname points to a unique file on a disk. In most cases, the Resource Manager will then transfer the resource's data to memory, and you receive a handle telling you where the data is located. After dereferencing the handle, you can do whatever you want with the data. If you change it,

A Parable

by Ross W. Lambert, Editor-in-Cognito

Once upon a time there was BLOAD... it was a nice little command, but fairly unintelligent. Apple II wizards everywhere used it to deposit subroutines, data, parameters and other goodies into their machines. These wizards spent hours finding little nooks and crannies in the II where they could squirrel away their stuff. Sometimes they ran out of nooks and crannies, though.

The fruit gods had compassion on the wizards, therefore, and bestowed new and powerful magic upon them - resources. But the wizards were baffled and befuddled. "What *is* a resource?", they cried. "And more importantly," they wailed, "WHERE is a resource?"

The fruit gods were strangely silent, but for good reason.

In time, the wizards began to play with the resources. It was not long before they discovered two important facts.

First, a resource could be whatever a wizard wanted it to be! The fruit gods decreed that there would be a bunch of "standard" resources with formats that would allow menu items, menu bars, buttons, window definitions, etc. to be put in resources and communicate clearly with new toolbox calls. The fruit gods described these in Appendix E of *The IIgs Toolbox Reference Volume III* (As things turned out, most of the wizards didn't need to know those details. A bunch of magic appliances started to appear on the market that let the wizards make those resources using pictures. Many, many wizards rejoiced.) In spite of standard resources,which were really just a convenience provided by the fruit gods, the wizards soon realized that they could put just about anything they wanted into the magical resource form.

Second, the wizards began to discover that, no matter what they put into a resource, they no longer had to worry about finding a nook or cranny for it. They could just do a little incantation and poof! Their resource would appear before them. Amazingly, they could even make more or bigger resources than would fit into memory all at once, and the resource magic would *still* make their resource appear on command.

This was powerful magic, indeed.

you can tell the Resource Manager that the data has changed with a _MarkResourceChange call. The Resource Manager will automatically update the resource on disk—perhaps not right away, but definitely before you quit your application. When you're done with data from the requested resource, you do nothing. The Resource Manager also manages the handle (i.e., creates/disposes of it) automatically.

There are system resources and user resources. System resources are predefined for a particular purpose, and have unique ID's associated with them. For example, there is a system resource dedicated for a pascal string. The pascal string resource has an type number of $8006 (all system resources start with $8xxx). In addition to the type, every resource has an ID number that you define in your program. This way, you can have more than one type of resource in each program.

There is nothing magical about system resources *(Editor: Oops. I blew that headline)*. If you wanted to, you could store graphics data in a pascal string resource; the Resource Manager wouldn't care one way or the other. It would still transfer your resource to memory as usual. System resources are merely a numbering scheme that standardizes the contents of resources.

However, standardization is important. By respecting system resource type numbers, you can be sure that your program will be compatible with other applications, such as resource editors. Similarly, future system software releases will likely extend the capabilities of the Resource Manager, and this could affect system resource handling.

The "user" resources are free-for-alls. That is, they can hold any kind of data you want. They are best suited for unique types of data needed by your program that cannot be satisfied by any of the standard system resource types.

"...standardization is important. By respecting system resource type numbers, you can be sure that your program will be compatible."

Applications of the Resource Fork

The typical resource file contains your compiled program in the data fork, and variable or static data in the resource fork. The advantage here is that you do not have to modify or re-compile your code to change a resource. Here are some specific applications:

Desktop Interface Data. This is the most often quoted application. All of your windows, buttons, alert strings, menu definitions, and so on could be contained in the resource fork. With an appropriate resource editor, you could create these items before you develop a single line of program code. In this way, you're not subject to the painful task of recompiling your code to move a window here or there or a button a few pixels left or right. Amazingly, we spend most of our programming efforts doing this.

Update Information. Rather than shipping your software with user manual addendum sheets, put the text in the resource fork, accessible by a menu selection in your main program. You won't have to re-compile the application for addendum changes, and other individuals (such as your tech support department) could update the file with a resource editor. If nothing else, you'll save paper and phone calls on common user problems.

Serial Number. Put a serial number (or create one when the program is first launched) in the resource fork. This way, every user can have a unique serial number that is displayed with the program, and becomes a permanent part of the individual copy.

Language Conversion. If you ship your software internationally, put all your ASCII strings in resources. Anyone can perform a language conversion of your software without needing to compile (or even have a copy of) your source code.

Graphics. Icons, cursors, and other graphic objects are a pain to modify, especially if they've been converted to source code. Using resources allows you to go back and do it graphically in half the time. *(Editor: It also permits a user with a resource editor to customize your graphics to suit themselves. Personally, I like this idea, but there are those folks that don't want the user messing with their program's guts in any fashion.)*

Pass Parameters. If your application consists of more than one executable file, you could pass information from one file to the next by modifying a common resource.

Digitized Sounds. Sounds take up an incredible amount of memory. Put them all in the resource fork, and they'll only be loaded by the Resource Manager when you program actually needs them. You may be able to easy the memory reuqirements for your application, or you might be able to include more sound files than were otherwise impossible.

Default Settings. Allow the user to customize his or her copy of your application by modifying default settings kept in a resource. When the program is launched again, the user gets customized defaults settings, not the factory settings.

User Modifications. Let's face it, just about everyone is a hacker to one extent or another. Allowing the end-user to modify menus and strings with a resource editor is not a bad thing, and may add a little extra sex appeal to your application.

New Ways to Solve Old Problems

Everyone likes to add graphics to an application. The problems associated with full screen graphics are they take up a lot of memory, and increase launch time (when embedded as source code in the application) or force you to use auxiliary files. I don't know about you, but I really dislike the latter. Unless the application is some piece of entertainment software with many support files in their own folder(s), I hate cluttering up subdirectories with lots of little support files. I never know who owns what file, and when I clean up my hard disk, and usually delete an important one.

If we put the graphic in a resource, we solve multiple problems at once. First off, the picture doesn't actually load when you launch the program; it only loads when the user chooses the menu item that invokes the resource. Secondly, being in a resource eliminates the need for an external support file. And third, we will use a packed (compressed) picture to lessen the disk space required to hold the application.

To unpack the graphic in the application, we will use a feature of the Resource Manager called a resource converter. Resource converters are handy things. They process the data from a resource in a special way before returning the handle in memory. In other words, the resource on disk and the resource in memory become two different things. In sample code, I included a resource converter that unpacks a super hi-res graphic image.

The application "Picture.Show" (listing one) is a complete desktop-based program that illustrates the Resource Manager in action. Most of the segments used to create menus and handle the main event loop are probably familiar to you, so I won't go into great detail on those. The program is organized into in the following segments:

❑ **MainCode** - Performs program initialization and handles the main TaskMaster event loop.

❑ **DoAbout** - A routine that displays a simple ABOUT window via the AlertWindow call.

❑ **DoPictureShow** - The routine that loads our picture resource into memory and displays it.

❑ **DoQuit** - Passes control to ShutDown when user selects the Quit menu.

❑ **PictConvert** - The Resource Converter code called by the Resource Manager.

❑ **InitMenus** - Sets up and displays the menu bar.

❑ **InitStuff** - Loads and starts up the tools.

❑ **ShutDown** - Shuts down all tools and quits.

❑ **MenuTables** - Menu definitions.

❑ **GlobalData** - Common data area and variables used by all other segments.

When Picture.Show is launched, a noticeably skimpy menu bar appears with just APPLE and FILE menus. Under the file selection is an item called SHOW PICTURE. When the user selects this item, the code in DoPictureShow is executed. Let's follow it through.

The first thing that happens is a _LoadResource call is made to get the graphics screen into memory. The Resource Manager returns a handle to us telling us where in memory the graphic is located. When you actually run this program, you'll notice that the first time you select this menu item, disk activity occurs. After that, the picture will appear, but no disk activity will take place. This is one of the beauties of the Resource Manager: you don't have to be concerned whether a resource is in memory or on disk because the resource manager automatically loads it if it is needed.

After the _LoadResource call, we lock and deference the handle returned by the Resource Manager to prevent it from moving around while we're using it. Then, we create a new window and copy the pixels of the graphic using a _PPToPort call. Because we want the user to view the picture for as long as he or she wants, we wait around for any event (which is likely to be a keypress or mouse click) by the _GetNextEvent call. When an event occurs, we dispose of our window, unlock the handle, and return to the main event loop.

You may be wondering how we magically unpacked this graphic without specific calls to _PackBytes or the routines to support it. Well, this is the good news about resource converters. The conversion is totally transparent to the _LoadResource routines. You simply call _LoadResource and the conversion is done before you get the handle. The bad news is that you have to write the conversion routine yourself. (Of course, once you've written a resource converter, you can use it in any of

your programs without having to recreate it from scratch.)

In listing one is a segment called PictConvert, which constitutes the picture unpacking routine. When we called _LoadResource, the Resource Manager called the PictConvert routine to find out how to convert the resource. In this case, _PackBytes was called to unpack the graphic image.

When the Resource Manager calls any resource converter, it passes a specific command that needs to be processed. If you've used custom controls in the past, you're familiar with the Control Manager's custom control DefProc procedure. This mechanism is very similar to converters. In this specific example, we only interpret the Resource Manager's request for READ command, which is our cue to read the resource data into memory and do whatever we have to in the conversion. Basically, PictConvert expands the size of the handle to make room for the unpacked version of the data, unpacks it, then resizes the handle accordingly.

Resource Converters are limited only by your imagination. Any time you need to process data in some way, the Resource Converter mechanism can do it. For example, suppose your resource converter processed its data through the Loader's _InitialLoad call. You could then save a code fragment in a resource, and the converter would do the relocation and loading of it for you. The dereferenced handle would be the starting address of another program! How about a converter that reads ProDOS directory blocks and transformed them into the standard ASCII strings used in a CATALOG command? Or a converter that changes an AppleWorks data base file to a random access text file? Just about anything is possible.

Creating Custom Resources

Picture.Show uses a custom, or user, resource. As such, we can't use a resource editor and need an additional program to "attach" or create the packed picture resource to the application's resource fork. Listing two, ResMaker, shows how to do this.

The Label "PictName" and "AppName" in listing two define the file names of our separate packed picture and application file, respectively. ResMaker searches the current directory for a file named "Res.Pict" which it assumes is a packed SHR screen. It then attaches that screen as a resource to the application named in the PictName label.

You can convert any paint file to the "PackBytes" ($C0/$0001) format using the shareware program SHRConvert or Roger Wagner's Graphics Exchange, or your paint program may save pictures directly in PackBytes format. In any case, the packed picture must be named Res.Pict for ResMaker to recognize it. Similarly, the application must be named "Picture.Show". Of course, you can change these two strings in ResMaker to anything you like. Note, however, that both strings are Class 1 GS/OS strings, meaning that they are proceeded by a word (16-bit) length field.

To keep things simple, ResMaker is an ORCA/M EXE fil and is executed under the Orca shell rather than a stand-alone program. To assemble Picture.Show and attach its packed picture resource, either type-in or create an exec file with the following commands:

```
asml ps.s
filetype ps S16
rename ps picture.show
resmaker
```

This assembles and links the program, changes the resulting file to a S16 type, renames it to Picture.Show, and runs ResMaker, attaching the resource.

Ripping into ResMaker

Let's see how ResMaker does its job. First, a few tools are started up, followed by a GS/OS OPEN call on the file Res.Pict. After the EOF (size of the file) is known, a _NewHandle call is made to allocate some memory, and the picture is loaded. Once in memory, a _CreateResourceFile call is made to convert Picture.Show to an extended file with a blank resource fork. Once that is done, the handle created is passed to the _AddResource call. This does all the work. The data (graphics) in memory are transferred to the file's resource fork and is assigned a type of $0020 and an ID of $00000001. These numbers are arbitrary, and you could use just about any type and ID values you like (as long as you don't step on system resource types).

The attributes we passed to _AddResource indicates an unlocked, purgeable block at level 3. This tells the Resource Manager that the resource can be purged from memory if the system needs the space. (The Resource Manager works best if you define all your

resources as purgeable.)

Next, we change the attributes of the resource with a _SetResourceAttr call. This tells the Resource Manager that a converter is needed for this resource. Note that you can't just set the Resource Converter bit in the attributes word when you make the _AddResource call. This would cause the Resource Manager to call the converter immediately, and since the converter (PictConvert routine) is actually located in our main application, the Resource Manager would generate an error indicating that a converter doesn't exist. Using the _SetResourceAttr call gets around this problem.

Once the new attributes are set, we close the resource fork of Picture.Show, dispose of the memory the picture occupied, and shutdown the shell application. The Resource Manager automatically saves all new resources to the resource fork generated by an _AddResource call.

ResMaker is a "quick and dirty" way to add a resource to an application. With some tweaks, you could get fancy and modify ResMaker to work with just about any resource. There is also very limited error checking and handling in ResMaker. For example, If you run ResMaker twice, that is, run it again from the shell after Picture.Show has already been processed by ResMaker, you will get a fatal error from the _AddResource call. This is because the resource you want to add already exists. The ambitious programmer is welcomed to modify and improve ResMaker, perhaps turning it into a mini resource editor!

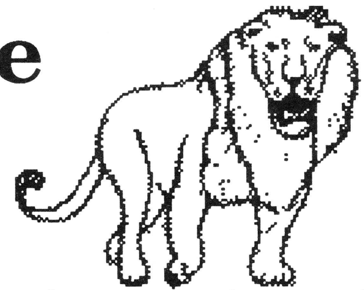
Where do we go from here?

We have barely scratched the surface in exploiting the Resource Manager's capabilities. Many of the calls not covered here (i.e., _AddAbsResource, _DetachResource, and _MatchResourceHandle) constitute a powerful set of tools that can be used to create unique functionality in your programs. And of course, you can still use resources to hold window parameters and button names. But this often-promoted use is minor compared to the full potential of this valuable new tool.

Listing 1:

```
************************************************
*
*
*
*
*                  Picture.Show
*
*
*
*       An example of how to use custom resources
*
*            By Joe Jaworski 4/19/90
*
*
*
*  ORCA/M Assembler 1.1
*
*
*
*******************************************************

           MCOPY        ps.mac
           KEEP         ps

MainCode   START
           USING        GlobalData

           PHK
           PLB
           CLD
           JSL          InitStuff        ;Init
toolsets
           JSL          InitMenus        ;Setup Men
           _InitCursor


; :::::::  TaskMaster Main Process Loop  ::::::::

TLoop      ENTRY

           WordSpace
           PushWord     #$FFFF     ;Accept all even
           PushLong     #TaskRec
           _TaskMaster

           PLA
           CMP        #wInMenuBar;Menu Item Selecte
           BNE          TLoop      ;Ignore all othe

; Menu Item Slected

           LDA          TaskData
           AND          #$00FF ;Create our Jump Tab
           ASL          A      ;Multiply by 2
           TAX
           JSR          (MenuJumps,X)    ;Go do it

TitleOff   PushWord     #0
           PushWord     TaskData+2
```

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

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

```
          _HiliteMenu           ;Highlighting off
          LDA           QuitFlag
          JEQ           TLoop
          JML           ShutDown;Quit application

; Menu Jump Table

MenuJumps DC            I'DoAbout'
          DC            I'DoPictureShow'
          DC            I'DoQuit'
          END


; ::::::::  Menu Selection Handlers   ::::::::::

;________________________________
;
; DoAbout
;
; Subroutine for the About Window.
;
;________________________________
;
DoAbout   START
          USING         GlobalData

          WordSpace                 ;Simple window
          PushWord      #1
          PushLong      #0
          PushLong      #AboutMsg
          _AlertWindow
          PLA                       ;Trash response
          RTS

AboutMsg  anop
          DC            C'43!'
          DC            H'01',C'S',I'$0001'
          DC            C'Picture Show'
          DC            H'01',C'S',I'$0000'
          DC            I1'CR'
          DC            C'By Joe Jaworski'
          DC            I1'CR',I1'CR'
          DC          C'Just a little program to
illustrate'
          DC            C' custom resources.'
          DC            C'!^#0'
          DC            I'0'

          END

;________________________________
;
; DoPictureShow
;
; Subroutine to display Packed Picture Resource.
;
```

```
;________________________________
;
DoPictureShow START
          USING        GlobalData

          LongSpace           ;Get Picture resourc
          PushWord     #rzMyID          ;Type
          PushLong     #rzPict          ;ID
          _LoadResource
          ErrorDeath   'LoadResource fatal'

          CopyLong     KeyHandle
          _Hlock                  ;Lock it
          Deref        KeyHandle ;Dereference it
          PushLong     <0
         PullLong  KeyPoint;Copy to SrcLocInfo re

          LongSpace
          PushLong     #KeyWParms   ;Open key wind
          _NewWindow
          PullLong     KeyWPtr      ;Save Grafport

; Copy Pixel Image to the window

          LongSpace
          _GetPort         ;Save current grafport
          PushLong     KeyWPtr;But leave on stack
          _SetPort

          PushLong     #KeyLocInfo;Source Location
          PushLong     #myrectkey ;Source rect
          PushWord     #0         ;X coord (none)
          PushWord     #0         ;Y coord (none)
          PushWord     #0        ;Default pen xfer
          _PPToPort

          WordSpace
          PushWord     #$FFFF
          PushWord     #0
          _FlushEvents               ;Clear queue
          PLA

NextEventH WordSpace
          PushWord     #$000E   ;Mouse/Keyboard Ma
          PushLong     #LocalHRec
          _GetNextEvent

          PLA                ;Get any non-null eve
          JEQ          NextEventH


AllDone   _SetPort                  ;Restore Grafport
          PushLong     KeyWPtr   ;Close it up
          _CloseWindow
          PushLong     KeyHandle      ;Unlock it
          _HUnLock
          RTS
```

```
; KeyEqu Window Parms and local scratch

KeyWPtr     DS            4     ;Window grafport ptr
KeyHandle   DS            4     ;Pic Resource Handle

KeyWParms   DC            I'KeyWEnd-KeyWParms'
            DC            I'%0000001000100100'
            DC            I4'0'
            DC            I4'0'
            DC            I'0,0,0,0'
            DC            I4'0'
            DC            I'0,0'
            DC            I'0,0'
            DC            I'0,0'
            DC            I'0,0'
            DC            I'0,0'
            DC            I4'0'
            DC            I'0'
            DC            I4'0'
            DC            I4'0'
            DC            I4'0'
            DC            I'13,0,200,640'
            DC            I4'-1'
            DC            I4'0'
KeyWEnd     anop

myrectkey   DC            I'10,0,200,640';Full Scrn
minus Menu Bar

KeyLocInfo  anop
            DC            I'128'   ;SrcLoc record
KeyPoint    DS            4  ;Pixel Ptr (filled-in)
            DC            I'160'  ;Width in bytes
            DC            I'0,0,200,640';BoundsRect

            END

;--------------------------------------
;
; DoQuit
;
; Subroutine for the "QUIT" Menu Item.
;
;--------------------------------------
;
DoQUIT      START
            USING         GlobalData

            INC           QuitFlag       ;Simply set our
Quit Flag
            RTS
            END

; ::::::::::  Support Routines ::::::::::::::::::
```

```
;--------------------------------------
;
; PictConvert
;
; This is the subroutine that converts the
application's packed
; resource into an unpacked picture. This routine
; is called only by the Resource Manager.
;
;--------------------------------------
;
PictConvert START      AuxCode
            USING      GlobalData

cvResRef    EQU        Linkin+0  ;Resrc Ref Word p
cvParms     EQU        Linkin+4  ;Cmmd Parms passe
cvCommand   EQU        Linkin+9  ;Converter Comman
LResult     EQU        Linkout+0 ;Spc for result
zpLocal     EQU        0         ;Local zero pg 4

            LINK      10,4 ;10 on stack,4 local va
            STZ       <LResult   ;NIL result alwa
            STZ        <LResult+2

            LDA        <cvCommand ;Check cmd type
            BEQ        Readcv     ;Go read It
            UNLINK
            RTL       ;Not a read command, ignore

; Exapnd the Handle Size

Readcv      PHB                   ;Allow abs addressin
            PHK
            PLB

            LDY        #16;Get Handle from resrc r
            LDA        [<cvResRef],Y
            STA        <zpLocal     ;To direct pag
            INY
            INY
            LDA        [<cvResRef],Y
            STA        <zpLocal+2

            LDY        #12         ;Get Size of Fil
            LDA        [<cvResRef],Y
            ADD     #$7D00;Expand by ScreenSize (32
            STA        RNewSize
            STZ        RNewSize+2
                        ;OK, never over 64K total

            PushLong   <zpLocal          ;UnLock it
            _HUnLock
            PushLong   RNewSize ;Resize it to actu
            PushLong   <zpLocal
            _SetHandleSize
```

```
         PushLong    <zpLocal  ;Lock it up again
         _HLock

; Update ReadParms in case it moved in memory
; (SetHandleSize can do that do you)

         LDA         [<zpLocal]
         LDY         #4   ;Start of Data buffer
(Class 1)
         STA         [<cvParms],Y
         STA         BuffStart
         LDY         #2
         LDA         [<zpLocal],Y   ;Do MSB, too
         LDY         #6
         STA         [<cvParms],Y
         STA         BuffStart+2

         PushLong   <cvParms;Issue READGS cmmnd
         PushWord    #ReadGS
         JSL         >GSOS

; Picture's in memory- go unpack it

         WordSpace
         PushLong    BuffStart;Start of packed
         LDY         #8
         LDA         [<cvParms],Y;Size of packed
         PHA
         STA         PackSize
         ADD         BuffStart;Create unpkd loc
         STA         UnPLoc
         LDA         BuffStart+2
         STA         UnPLoc+2
         PushLong    #UnPLoc ;Ptr to unpack Ptr
         LDA         #$7D00
         STA         UnPSize
         PushLong    #UnPSize;Size unpking area
         _UnPackBytes
         PLA                 ;Trash actuals

; Move it up in memory
         LDA         BuffStart+2;Src(unpkd loc)
         PHA
         LDY         #8
         LDA         [<cvParms],Y;Size of pkd
         ADD         BuffStart   ;offset into
    unpacked portion
         PHA
         PushLong    BuffStart    ;Destination
         PushLong    #$7D00       ;Length
         _BlockMove

ExitConv anop
         LDA         #0                 ;Clears
carry flag
         PLB
         UNLINK
```

```
         RTL

; Local storage

BuffStart  DS          4     ;Memory Start Pointer
PackSize   DS          4     ;Size of packed porti
RNewSize   DS          4     ;New Size of Handle
UnPloc     DS          4     ;Unpacked location
UnPSize    DS          2     ;Size of unpked porti

           END


;-----------------------------------
;
; InitMenus
;
; Creates our list of menu items.
;
;-----------------------------------
;
InitMenus  START       AuxCode
           USING       MenuTables

; Insert my menu lists/items

           PushLong    #0
           PushLong    #Menu2            ;File
           _NewMenu
           PushWord    #0
           _InsertMenu

           PushLong    #0
           PushLong    #Menu1            ;Apple
           _NewMenu
           PushWord    #0
           _InsertMenu

           PushWord    #1          ;Apple Menu ID=1
           _FixAppleMenu
           WordSpace               ;Adjust internal
           _FixMenuBar
           PLA
           _DrawMenuBar            ;...And draw it
           RTL
           END


;-----------------------------------
;
; InitStuff
;
; Init the Tool Sets and program global variables.
;
;-----------------------------------
;
```

```
InitStuff  START        AuxCode
           USING        GlobalData

TableSize  EQU          TTend-ToolArray;Size of our
tool array

           _TLStartUp          ;Do it by the book
           _IMStartUp
           _TextStartUp
           _ADBStartUp
           WordSpace           ;space for ID
           _MMStartUp          ;Memory Mgr now up
           PLA                 ;Get Master ID
           STA          >MasterID

           LongSpace           ;Space for SSRecord
           LDA          >MasterID
           ORA          #$0100;Private Tool ID num
           PHA
           PushWord     #0       ;Pass Pointer Verb
           PushLong     #ToolRecord
           _StartUpTools
           ErrorDeath   'Cant load needed tools'

           PullLong     >SSReference;save reference
                             ;for quitting


; Install Converter for our picture

           PushLong     #PictConvert ;Our Routine
           PushWord     #rzMyID
           PushWord     #1            ;Log in
           _ResourceConverter
           ErrorDeath   'Converter Login failure'
           RTL

; Tool Record is here

ToolRecord ENTRY
           DC           I'0'
           DC           I'$4080';640 Mode, FastPort
                              ;no shadowing
           DS           2        ;Resource ID space
DPHandle   DS           4        ;Direct Pg Hndl spc
           DC         I'TableSize/4';# of Toolsets
                               ;in Table

ToolArray  anop
           DC           I'3,$0000' ;Misc Tools
           DC           I'4,$0301' ;QuickDraw (5.02
check)
           DC           I'18,$0000';QuickDraw Aux
           DC           I'6,$0000' ;Event Manager
           DC           I'14,$0000';Window Manager
           DC           I'16,$0000';Cntrl Manager
           DC           I'15,$0000';Menu Manager
           DC           I'20,$0000'   ;Line Edit
           DC           I'21,$0000'   ;Dialog Manag
           DC           I'23,$0000'   ;Standard Fil
           DC           I'22,$0000'   ;Scrap Manage
           DC           I'5,$0000'    ;Desk Manager
           DC           I'28,$0000'   ;List Manager
           DC           I'27,$0000'   ;Font Manager
TTEnd      anop

           END


;--------------------------------
;
; ShutDown
;
; Shuts down all the tools, releases memory,
; and gracefully quits back to launcher.
;
;--------------------------------
;
ShutDown   START        AuxCode
           USING        GlobalData

           PushWord  #0 ;Shut down array-based too
           PushLong     >SSReference
           _ShutDownTools

           PushWord     >MasterID  ;Dispose of me
           _MMShutDown
           _ADBShutDown


           _TextShutDown        ;All the rest, too
           _IMShutDown
           _TLShutDown

           _QuitGS    QuitParms ;...and shut 'er do
           BRK          $F0

; Quit parm table

QuitParms  DC           I'2'      ;pcount
           DC           I4'0'     ;no next applicati
           DC           I'$0000' ;can't restart

           END

;------------------------------
;
; MenuTables
;
; Used by InitMenus.
;
;------------------------------
;
MenuTables DATA         AuxCode
```

```
            USING       GlobalData

Menu1       anop
         DC      C'>>@\H',I'AppleMID',C'X',I1'CR'
         DC      C'##About Picture
Show...\H',I'AboutMID',I1'CR'

            DC          C'##-\N400D',I1'CR'
            DC          C'.'

Menu2       anop
            DC    C'>>  File   \H',I'FileMID',I1'CR'
            DC    C'##Show
Picture\VH',I'ShowPMID',I1'CR'
            DC
C'##Quit\*QqH',I'QuitMID',I1'CR'
            DC          C'.'

            END

;----------------------------------
;
; GlobalData
;
; Mostly equates, Mostly used by all.
;
;----------------------------------
;
GlobalData DATA

rzMyID      GEQU     $0020;My private user Res ID
rzPict      GEQU     $00000001;Type for packed pict
GSOS        GEQU     $E100B0;Stack-based entry point
ReadGS      GEQU     $2012  ;GS/OS Read command
CR          GEQU        $0D ;You guessed it

; Global variables

MasterID    DS          2  ;Mstr ID from MMStartUp
SSReference DS          4  ;Start/Stop tool ref
QuitFlag    DS          2  ;Quit Flag
temp        DS          16 ;Temp for anyone to use
anyhow

LocalHRec   anop
lWhat        DS          2      ;Event Code
lMessage     DS          4      ;Event Result
lWhen        DS          4      ;Ticks since Startup
lWhere       DS          4      ;Mouse loc (global)
lModifiers   DS          2      ;Status of Modifier

; Menu Bar References

AboutMID    GEQU         256    ;Menu Item ID's
ShowPMID    GEQU         257
QuitMID     GEQU        258

FileMID     GEQU        2             ;Menu ID's
AppleMID    GEQU        1
; Event Management

TaskRec     anop
What        DS          2        ;Event Code
Message     DS          4        ;Event Result
When        DS          4        ;Ticks since
Startup
Where       DS          4        ;Mouse loc (globa
Modifiers   DS          2        ;Status of Modifi
TaskData    DS          4        ;TaskMaster Data
TaskMask    DC          I4'$001FFFFF'    ;Mask Bit
LastClTick  DS          4       ;Last Click
TickCount
ClickCount  DS          2       ;Click Count
TaskData2   DS          4       ;Control Handle
TaskData3   DS          4       ;part code mous up
dn
TaskData4   DS          4       ;Control ID
LastClPt    DS          2       ;Last Click Point
            DS          2       ;Last Click Point
wInMenuBar  GEQU        $0011   ;In Menu Bar

            END
```

Listing two:

```
*****************************************************
*
*
*
*
*                      Res.Maker
*
*
*
*              A custom resource maker.
*
*               By Joe Jaworski 4/19/90
*
*
*
*  ORCA/M Assembler 1.1
*
*
*
*****************************************************

            KEEP        RESMAKER
            MCOPY       RES.mac
```

```
MainCode   START

rzMyID     GEQU        $0020 ;My private Res ID
rzPict     GEQU        $00000001;Type for packed
pict

           PHK
           PLB
           CLD

; Initialize the few tools we need

           _TLStartUp          ;Tool Locator
           WordSpace
           _MMStartUp          ;Memory Manager
           PLA
           STA         MasterID
           _MTStartUp          ;Misc. Tools
           PushWord    MasterID ;Resource Mgr
           _ResourceStartUp

           _OpenGS     opParms;Open the pic file
           ErrorDeath  'Cant open file'

           LongSpace           ;Allocate memory
           PushLong    opEOF ;Exact file size
           LDA         MasterID
           ORA         #$0200  ;make it private
           PHA
           PEA         $C000;lockd,fixd,anywhere
           PushLong    #0
           _NewHandle
           ErrorDeath  'NewHandle failure'

           PullLong    PcHandle  ;Recover handle
           Deref       PcHandle  ;Dereference
           Copy4       <0,RBuffer;Copy buffer ptr
           LDA         opRefNum  ;Copy Refnum's
           STA         rdRefnum
           STA         Clrefnum
           LDA         opEOF     ;Copy file size
           STA         RCount
           STZ         RCount+2
           _ReadGS     rdParms   ;Read the file
           ErrorDeath  'Read Failure'
           _CloseGS    ClParms   ;Close it

; Create the Resource fork

CreFork    PushLong    #$0000       ;No Auxtype
           PushWord    #$0000       ;No Filetype
           PushWord    #$0000       ;No Access
           PushLong    #AppName     ;Filename
           _CreateResourceFile

; Open the Resource fork

           WordSpace
           PushWord    #$0003          ;Access
           PushLong    #0              ;No header
           PushLong    #AppName        ;Filename
           _OpenResourceFile
           ErrorDeath  'OpenResource fatal'

           Pullword    ResFileID    ;Recover Res ID

; Attach the Picture file (now in memory)

           PushLong    PcHandle   ;Resource Handle
           PushWord    #$0300     ;Purge level 3
           PushWord    #RzMyID    ;Resource ID
           PushLong    #rzPict    ;Resource type
           _AddResource
           ErrorDeath  'AddResource fatal'

           PushWord    #$0B00    ;Set Converter bit
           PushWord    #RzMyID
           PushLong    #rzPict
           _SetResourceAttr

; Done- Quit and return to the shell

           PushWord    ResFileID  ;Close it up
           _CloseResourceFile
           PushLong    PcHandle   ;Free memory
           _DisposeHandle

           _ResourceShutDown
           _MTShutDown               ;Misc Tools
           PushWord    MasterID      ;Dispose of me
           _MMShutDown
           _TLShutDown
           RTL          ;Use QUIT here for S16 file


; Scratchpad and variables

MasterID   DS          2          ;Master ID from
MMStartUp
ResFileID  DS          2   ;Resource ID returned
PcHandle   DS          4   ;Picture-in-memory Handl

AppName    STRL        'Picture.Show' ;Application
Name (Class 1)
PictName   STRL        'Res.Pict'     ;Picture File
name (Class 1)

; OPEN Parms for Picture File

opParms    DC          I'12'           ;PCount
opRefnum   DS          2            ;Returned Refnum
           DC          I4'PictName';Ptr to Pathname
           DC          I'$0001' ;Access (Read Only)
```

```
         DC        I'0'   ;Fork (Data)
         DS        2      ;Returned Access
         DS        2      ;Filetype
         DS        4      ;Aux Type
         DS        2      ;Storage type
         DS        8      ;Creation Date
         DS        8      ;Modification Date
         DS        4      ;Option List Pointer
opEOF    DS        4      ;File Size


; READ Parms for Picture File

rdParms     DC        I'4' ;PCount
rdRefNum    DS        2  ;Refnum (filled-in)
RBuffer     DS        4  ;Data Buffer (filled-in)
RCount      DS        4  ;Number of bytes to read
            DS        4  ;Returned transfer count


; CLOSE Parms for Picture File

ClParms      DC        I'1';PCount
ClRefNum     DS        2  ;Refnum (filled-in)

             END
```


Meet Other Apple II Developers!

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

Attend Apple's IIgs College

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

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

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

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

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

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

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

WE WANT YOUR BEST!

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

Let us take a look at your work! We are the publisher of Softdisk™ and Softdisk G-S,™ a pair of monthly software collections sold by subscription, on newsstands and in bookstores everywhere. We are looking for top-notch Apple software. We respond promptly, pay well, and are actually fun to work with!

What have you got to lose? Nothing! You could see your software published and earn cold, hard cash. Send your best software to:

Jay Wilbur
c/o Softdisk Publishing, Inc.
606 Common St. Dept. ES, Shreveport, LA 71101
GEnie: JJJ / America Online: Cycles

Here's a short list of the types of programs that will put a gleam in our eyes (and money in your pocket)! For more details, call...
Jay Wilbur
(318) 221-5134

APPLICATIONS
UTILITIES
EDUCATION
ENTERTAINMENT
GRAPHICS
FONTS
DESK ACCESSORIES,
INITS, CDEVS, ETC.

The Pascal Protocol and You

A Modest Proposal

by Ross W. Lambert

Part I - The Wherefore and the Why

This may be a ZBasic column, but this month everybody should read at least the first part of it. I'd like to make a suggestion for the organization of assembly language routines we print so as to increase their usability by high level language programmers of all stripes (and perhaps even other assembly junkies). Though this article will deal with things in an 8 bit context, the idea is even more important for IIgs code.

As a preface to it all, it is important to understand that a compiler is really a general purpose machine code generator. Whether you are using ZBasic on a 128K IIe or Orca C on a 2 megabyte IIgs, the compiler cannot always generate the "absolutely positively" most efficient code possible to accomplish your task. Thus, even in this modern age of speedy CPUs and acres of RAM, custom assembly language is still sometimes the best means to any given end. (A momentary aside: I believe that hardware technology is presently outpacing software technology. The efficiency and speed of any given piece of software is at least as dependent on the software as the hardware. In fact, *MacTutor* ran a piece a few months ago pointing out the truly inefficient code pumped out by several of the popular Mac compilers. My point: an optimized IIgs program can outperform many Mac applications in several key areas. More importantly, an optimized IIgs program can make other IIgs programs look like they're mired in cement.)

Back at the ranch, let's look at the difficulty we can encounter mixing high level code (in this case, ZBasic) with assembly language. My case in point shall be Tom Hoover's excellent AppleWorks™-style input routine from the May, '90 issue.

With a few bytes worth of phenagling on my part, Tom's routine assembled to about 556 bytes. A functionally identical routine Ariel distributes with ProTools tops out at a whopping 4000+ bytes, the reason being that it is all compiled ZBasic source code. This highlights yet another advantage of assembly language - size. With no extranneous JSRs, range checks, etc., assembly code can be relatively tiny compared to its compiled counterpart.

Tom's routine began by assuming that the X and A registers were "pre-loaded" with the prompt character and the maximum length. This is a fairly common practice amongst those who live and die in the rarefied air of the all-assembly language world. Unfortunately, most high level languages do not have commands which load the registers of the CPU with a value. That is a pretty low level function, hence incorporating Tom's routine with ZBasic (or a similarly constructed IIgs assembly routine with C or Pascal or anything else) is problematic.

"Everybody should read at least the first part of this..."

Enter the Pascal calling protocol (gasp!). Before 9,000 assembly junkies have me drawn and quartered, let me point out that I freely admit that in some instances the Pascal protocol is too slow and/or cumbersome. An

animation routine would die of old age if it had to retreive parameters this way while in a loop. That is why we are not absolutely mandating it. Conventions are useful only as long as they are useful. We are making but a modest proposal in hopes that more of our assembly listings will be more accessible to more people.

Wha Issit?

So, I already hear many of you asking, "Whazza Pascal protocol?"

Apple IIgs programmers may not know it by name, but they already use the Pascal protocol every time they make a toolbox call. Well, almost every time. The Pascal protocol is simply a mechanism for passing parameters (such as rectangle boundaries, string lengths, etc.) wherein you push the little buggers onto the stack and JSR (or JSL) to the subroutine in question. It is called the Pascal protocol because some guy named Pascal invented it. (I joketh - its conventions are those used by the Pascal language for passing paraemters.)

At this point it makes the most sense to start telling you about how to implement the Pascal calling protocol from ZBasic or another high level language. Instead I'm going to digress and tell assembly language programmers how to receive parameters this way (so they don't have to read any farther). The rest of you can skip on to **Part II, You can bet your LIFO.**

Pass the Parms, Please

As I mentioned earlier, Tom's original program expected the two parameters to be passed in the X register and the accumulator. To make both his code and your own assembly language creations Pascal protocol compatible:

1 - Pull the return address off the stack and store it somewhere temporarily (the 65C02 and 65816 opcodes PLY and PLX are handy for this)

2 - Pull your parameters. Obviously, you must remember that you will be pulling them down in the opposite order the caller pushed them.

3- Push any parameters you want to return to the caller. Remember that your caller will be pulling them off in the opposite order you push them.

4 - Push the return address back on top of the stack. Watch out for the order of things, here, too. The high byte (or word) goes on first, followed immediately thereafter by the low byte (or word). I always messed this up until I remembered one simple mnemonic rule: the **l**ow byte should go on **l**ast. I've also discovered that the seven dwarves had it right, too. ("Hi Lo, Hi Lo, we push the parms and go...")

For Tom's input program, the revised opening sequence looks like this:

Listing 1 - Pulling Parms from the Stack

```
94 GetStr   ent     ;Merlinese for making var
                         avail to other modules
95          sty    yTemp  ;to preserve the Y regist
96          plx           ;pull return addr off sta
97          ply           ; and store in X and Y
98
99          pla       ;top parm is max length
100         sta    MaxLength
101         pla       ;next parm is prompt characte
102
103         phy       ;push return addr back on sta
104         phx     ; (in opposite order pulled!)
105
106
```

If you are writing for 6502s, you can temporarily store the return address in the registers by using a PLA,TAY, TYA,PHA sequence. Since the 128K version of ZBasic only runs on enhanced IIe's, IIc's, or IIgs's, it was fair game for me to expect a 65C02 or later chip. By the way, if necessary, you can also store the return address at a specific memory location and push it back on the stack at any time.

This is exactly what you'd want to do if you need to pass any parameters *back* to the caller; push the parms on the stack *before* redepositing the return address on top of the stack. Like so:

Listing 2 - Returning Parms to the Caller

```
900      lda Parm1;push vals to return to caller...
901      pha
902      lda Parm2
903      pha
904      lda Return+1;THEN push return addr, high
byte FIRST!
905      pha
906      lda Return
907      pha
908      rts               ;RTS takes us home
```

Part II. You can bet your LIFO

When programmers say "THE" stack we mean something a little different than when we say "A" stack. Let's look at the general case first.

A stack is simply a data structure where data items are stored in order, one right on top of the other (see Figure 1). A little beast called the "stack pointer" keeps track of the top of the stack. Since there is usually a limit to the size of the stack, if software tries to put too much data on it we get what is called a "stack overflow" (does that have a familiar ring to it?) Furthermore and most importantly, the type of stack we want to talk about is arranged such that the first item removed is the last item deposited. The acronym for this arrangement is LIFO, for **L**ast **I**n **F**irst **O**ut.

Roger Wagner's restaurant analogy is the best one I've heard; imagine the spring loaded stack of plates common to so many Denny's, International House of Pancakes, etc. If the busboy puts 30 clean plates on the plate stack, all the plates below get squished down somewhere below the counter. When the waitress comes by she grabs the top three or four plates, thereby removing the last items deposited. The plates immediately below spring up, anxiously awaiting their moment of glory. This is just like a LIFO stack.

There are other kinds of stacks (FIFO, FIDO, HARPO, GROUCHO), but we'll leave them for another day.

You can create your own stacks for your own private uses, but in an Apple II (and most other computers) there is a (LIFO) system stack that is pretty much accessible by anyone at anytime. This is THE stack, and we must be careful what we do to it or we can bring the entire system to its knees.

A real world example

Let's use the Pascal calling protocol as an example. If I try to write a ZBasic function that pushes parameters onto the stack (which I did), I immediately run into a problem (which I did). Whenever a program needs to temporarily transfer control elsewhere, which is the case with a function or subroutine, it pushes the "home" or "return" address onto the stack (minus one if you want to be picky) and then starts executing at the new location. When an RTS, RETURN, END FN, or some similar statement is encountered, the machine yanks the return address down off the stack and jumps back to that place in memory. If you're a really tricky and gutsy person, you can change the return address on the stack on the fly and have a program return to some place other than whence it came. This is not for the faint of heart and is not germaine to the discussion at hand, although it is sorta fun.

Back to the point, if my ZBasic function merely pushes parameters on the stack, it will make the top items of the stack the parameters just pushed and NOT the return address. This is not good. It made my program jump into the midst of an array where it died a miserable death. This is why we must be careful with the stack. Under certain conditions it determines program flow.

"Both halves of a 128K machine running under ZBasic are therefore connected by the same stack and zero page."

To create a ZBasic function that pushes a value onto the stack, I had to "lift up" the return address for the function itself, deposit my values, and then set the return address back down on top. By doing business in this manner the function didn't self-destruct when it finished.

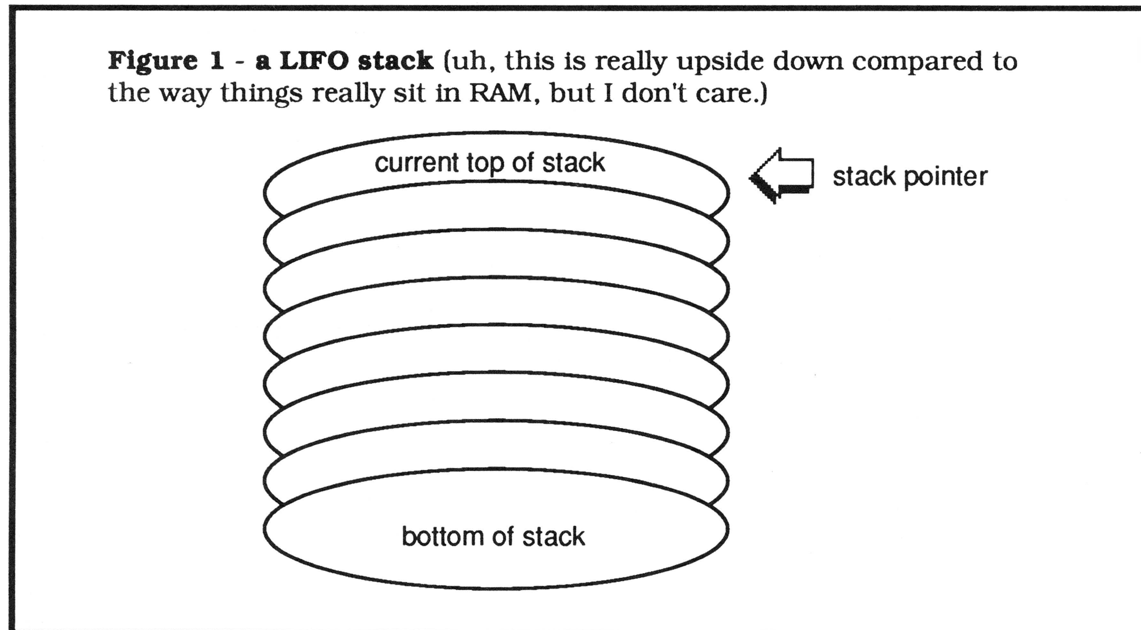


Figure 1 - a LIFO stack (uh, this is really upside down compared to the way things really sit in RAM, but I don't care.)

Since ZBasic (and most high level languages) don't have register oriented commands, it was imperative that I invoke some in-line assembly via the MACHLG statement. In doing this I discovered some things that will be of great interest to those of you who like to mix assembly language and ZBasic.

I got a secret...

First, the "dual bank" paradox of the 128K version of ZBasic makes a lot of assembly types dizzy. The key thing to remember is that the variable space is in main memory and the program space is in auxiliary memory. If you try to write in-line assembly with a MACHLG statement, the code is part of the ZBasic program proper and will be running in **aux mem.** The confusion stems from the fact that if you BLOAD an assembly routine into a buffer created in the variable space, it will be humming merrily along in **main memory**.

Most of us use the BLOAD technique to pull in external assembly language functions, so how in the world do we get a program running in aux mem to pass parameters to a program running in main mem?

A stickey wicket - until I found out that ZBasic leaves main memory zero page and stack active *at all times while your application is running.* Keep in mind that the whole environment is fluid when you drop back into the editor; things are moving around all over the place.

Both halves of a 128K machine running under ZBasic are therefore connected by the same stack and zero page. With that out of the way, it was an easy task to send any arbitrary variable to a zero page location and push it on the stack for use by someone else's subroutine.

The true byte scroungers amongst you will have realized by now that one important implication of all this is that we have uncovered a heretofore unknown 512 byte block for programs or data living in aux mem (i.e. the aux mem stack and zero page, $00-$1FF). Though tricky to use, it can really be nice to have it available (especially since zero page memory is so precious!).

Here's what FN PushByte looks like with the assembly opcodes along side the Zbasic code:

Listing 3 - FN PushByte

```
LONG FN PushByte (Byte)
  Zpg = PEEK WORD (6):REM  save zero page stash
  POKE ZPtr,Byte     :REM  store in zero pg scrat
  MACHLG &FA         :REM  plx  ;pull return addr
  MACHLG &7A         :REM  ply
  :
  MACHLG &A5,&06     :REM  lda $06 ;get byte into
  MACHLG &48         :REM  pha     ;push it
  :
```

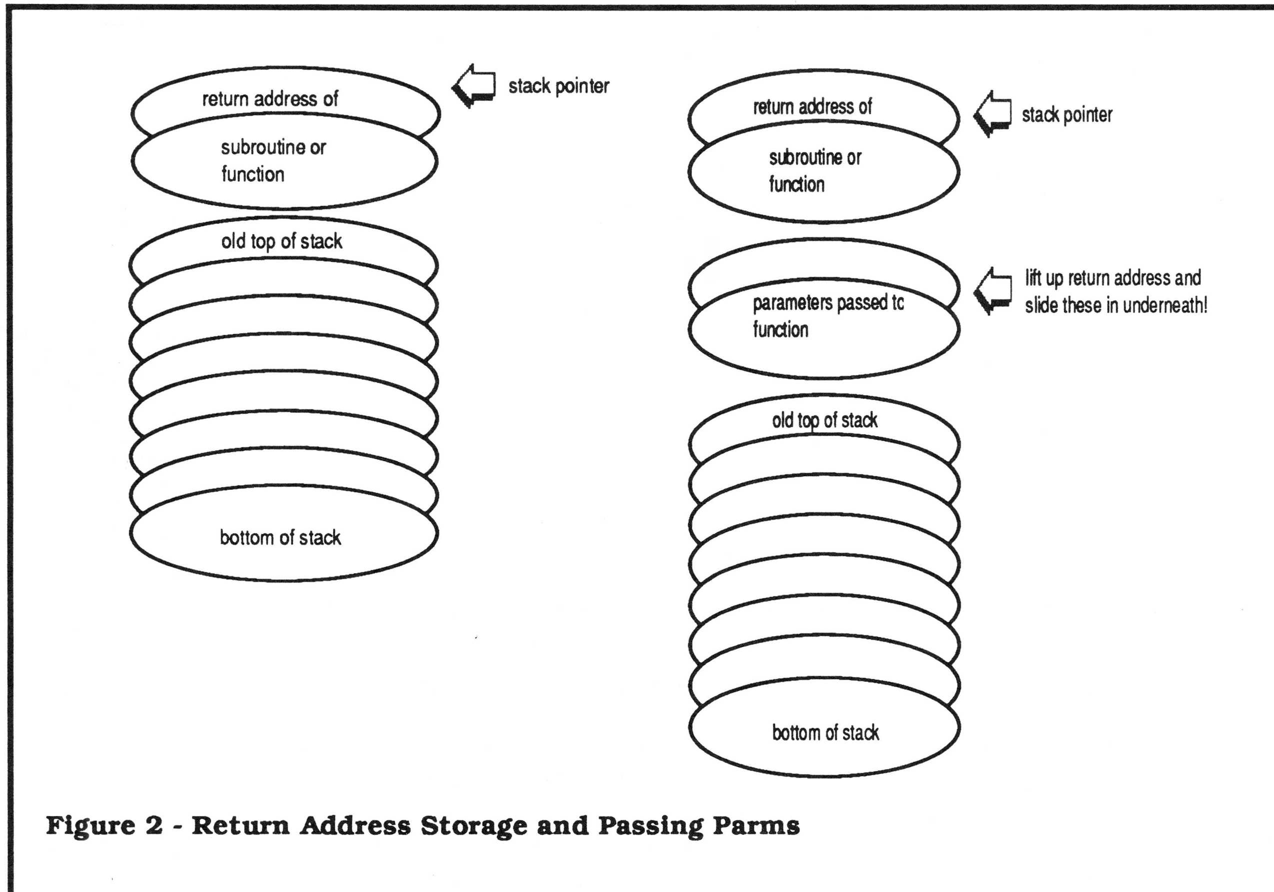


Figure 2 - Return Address Storage and Passing Parms

```
  MACHLG &5A       :REM  phy ;restore ret addr
  MACHLG &DA       :REM  phx
  POKE WORD 6,ZPg :REM  restore zero pg
END FN
:
```

You'll notice that I saved and restored the zero page locations used. This may not be strictly necessary since I believe the locations involved, bytes 6 and 7, are primarily scratch space. But I seem to recall Greg Branche (the primary developer of the ProDOS version of the language who now works for Apple, Inc.) reminding folks on GEnie to "...leave my zero page memory alone". I think ZBasic makes extremely heavy use of the zero page, so it is safest to clean up any messes made there. Removing the lines that save and restore these bytes would speed up the function considerably if you need it, but you do so at your own risk.

Since some of the routines we print may need to pass a value back to the caller, we also need a mechanism for retrieving values off the stack. Enter FN PullByte:

Listing 4 - FN PullByte

```
LONG FN PullByte (DestAddr)
  Zpg = PEEK WORD (6)
  POKE WORD ZPtr, DestAddr
  MACHLG &FA     : REM  plx  ;pull off return addr
  MACHLG &7A     : REM  ply
  :
  MACHLG &68     : REM  pla  ;pull byte off stack
  MACHLG &5A     : REM  phy  ;Y part of ret addr
  MACHLG &A0,&00: REM  ldy #0 ;init y register
  MACHLG &91,&06: REM  sta (06),y ;indexed indire
               : REM              ;to dest
  :
  MACHLG &DA     : REM  phx ;push X part of ret ad
  POKE WORD 6,Zpg
END FN
:
```

It is certainly possible to call FN PushByte and FN PullByte repeatedly to push and pull word length (2 byte) values on and off the stack, but as it happens this is significantly slower than a customized word length

function. Since many parms are word length, and since speed may be important, I therefore offer FN PushWord and FN PullWord

Listing 5 - FN PushWord and FN PullWord

```
 LONG FN PushWord (Pointer):REM gimme addr of
                           :REM word-len val
   Zpg = PEEK WORD (6)
   POKE WORD ZPtr,Pointer :REM  store pointer to
                          :REM  source in zero pg
   MACHLG &FA      :REM  plx ;pull ret addr
   MACHLG &7A      :REM  ply
   :
   MACHLG &A5,&07  :REM lda $07;hi byte 1st!
   MACHLG &48      :REM pha   ;push on stack
   MACHLG &A5,&06  :REM lda $06 ;get lo byte
   MACHLG &48      :REM pha  ;& push on stack
   :
   MACHLG &5A      :REM phy ;restore ret addr
   MACHLG &DA      :REM  phx
   POKE WORD 6,Zpg
 END FN
 :
 LONG FN PullWord (DestAddr)
   Zpg = PEEK WORD (6)
   Zpg2 = PEEK(8)  :REM  we need one more zpg
                     :REM byte for scratch spc
   POKE WORD ZPtr, DestAddr
   MACHLG &FA      :REM plx ;pull return addr
   MACHLG &7A      :REM ply
   MACHLG &84,&08  :REM sta $08;stuff 2 scrch
   :
   MACHLG &68      :REM pla ;pull lo byte
   MACHLG &A0,&00  :REM  ldy #0 ;init y reg
   MACHLG &91,&06  :REM  sta (06),y
                   :REM ;indxd indrct to dest
   MACHLG &68      :REM  pla    ;pull hi byte
   MACHLG &CB      :REM  iny  ;inc Y register
   MACHLG &91,&06  :REM  sta (06),y
                   :REM ;indxd indrct to dest
   :
   MACHLG &A4,&08  :REM ldy $06 ;retrieve Y
   MACHLG &5A      :REM phy ;restore ret addr
   MACHLG &DA      :REM phx
   POKE WORD 6,Zpg :REM rest. 0 pg bytes 6-8
   POKE 8,Zpg2
END FN
 :
```

If you think back to the early parts of this article, the issue that sparked it all was that I wanted to convert Tom Hoover's AppleWorks-style input routine such that it could be accessed by ZBasic programs. The program listing below accomplishes that goal, but it assumes that you've already typed in FN PushWord, FN PushByte, FN PullWord, and FN PullByte. It also assumes that you've made the beginning of Tom's program comply with the Pascal calling protocol. If you have the Merlin assembler, just make the changes I did in the opening lines of code of the program (c.f. Listing 1). By far and away the best idea with the least amount of work is to buy the June disk (if you don't already subscribe). I'll have the object code and the revised source in the ZBasic folder. Individual disks are $8.00, but if you are really only interested in that one file, send us a self-addressed stamped envelope and a blank, formatted disk, and we'll copy the file over you and return your stuff. Please allow 3 - 4 weeks.

There is one other important thing to note: Tom's original code was a rel module, meaning that Merlin would resolve the address references when it was linked with all the other modules in a program. This a really neat way to put together assembly code, but it leaves high level languages out in the cold. I remedied that by ORGing it to $9DCC and making the **first line of my ZBasic DIM statements** look like this:

```
DIM InputLoc,InputLoc(280)
```

The InputLoc() array is our buffer for the AppleWorks input routine, and InputLoc is going to eventually tell us where the buffer starts (via the VARPTR statement). In this fashion you can determine where your assembly code is going to eventually live and then ORG it appropriately. The only caveats are that you must reORG and reassemble your assembly code if 1) you change the position of the DIM statement, or 2) you reconfigure ZBasic such that you allow more open files. Such a reconfiguration changes the variable space and thus the position of your buffer. Fixed position assembly routines will choke if they're not ORG'ed to run at the right spot.

FN SuperInput$

I call Tom's input routine FN SuperInput$. The calling syntax is a tad odd, but it serves a purpose. Here's what it looks like:

```
FN SuperInput$(X,Y,MaxLength,PromptChar,DefPtr)
```

The first parameters are easy; they are the horizontal and vertical positions of the beginning of the input line

in text screen coordinates. As Tom pointed out last month, positioning the input is important for long input lines since word wrapping is not supported.

The third parm is the longest input you are willing to accept, and the fourth is the ASCII value of the character you would like for a prompt. A typical AppleWorks prompt is the greater-than sign (">"). Note that this is not the same as the cursor character. The prompt character sits immediately to the left of the input line.

The last parameter, DefPtr, is the *address* of the default string. If you have no default string, pass the address of a null string. You can derive the address by using VARPTR, of course.

Why pass the address when you could pass the actual string? Well, for one thing it saves a little memory - there is no extra string required for use by the function itself. For another thing we must move the results of FN SuperInput$ into a variable somewhere, and using the original string saves a step and just generally makes things go a little faster.

An important "gotcha" here: the string whose address you pass to the function will end up holding the result of the input. You must make certain that this string is DIMensioned to a length one byte greater than the MaxLength you specify for the input. The extra byte is for the leading length byte, by the way. If your target string is not sufficiently DIMmed, the input routine could overwrite other variables.

Conclusion

I am not too proud to say that Tom's AppleWorks-style input routine is a better alternative than the FN SmartInput$ we distribute with ProTools. I've been meaning to write a routine like this in assembly for a long time, but since he beat me to it I'm glad for the time saved! You'll notice that FN SuperInput$ is suprisingly short, even if you count the FN BLOAD (necessary to read it in). This code will give you slightly better functionality, greater speed, and more program space since it is 3.5K smaller.

I hope you ZFans have as much fun with it as I have!

Next Month: Local Variables!!! Stay tuned.

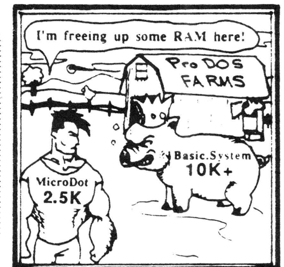
Listing 6

```
REM ------------------------------------
REM  AppleWorks-Style Line Input
REM  for ZBasic
REM
REM  All hard work done by
REM  Tom Hoover
REM  Copyright (C) 1990
REM  Ariel Publishing
REM  Some Rights Reserved
REM
REM  Pascal calling protocol for
REM  ZBasic by Ross W. Lambert
REM
REM ------------------------------------
:
REM Default variable type is integer, expressions
REM optimized to integer, & convert to case is NO.
:
REM ------------------------------
REM      DIMension stuff
REM ------------------------------
:
DIM InputLoc,InputLoc(280)
DIM 65 Path$, 2 Prompt$, 40 TheString$
:
REM ------------------------------
REM    A few lonely equates
REM ------------------------------
:
InputLoc = VARPTR(InputLoc(0)) : REM calc BLOAD ad
PgMovLoc = 780
Prompt$ = ">"    : REM like AppleWorks
OurCH = &057B    : REM horz cursr pos on 80 col sc
ZPtr = 6
InBuffer = &200
:
REM ---------------
REM     Define Functions
REM ---------------
:
:
REM The FN PushByte function from article must be
REM included here!!!!!  (Listing 3)
:
:
REM This is straight from ZBasic disk but properly
REM modified for the 128K version
:
LONG FN BLOAD (Path$,FileNum,Address,Length)
  Buffer = &AC00 - (FileNum * &400)
  POKE WORD &1F01, VARPTR(Path$)
  LONG IF Address = 0
    POKE &1F00,10
    MACHLG &A9,&C4,&20,&0865
```

```
    Address = PEEK WORD (&1F05)
  END IF
  IF Length = 0 THEN Length = &FFFF
  POKE WORD &1F03, Buffer
  POKE &1F00,3
  MACHLG &A9,&C8,&20,&0865
  POKE &1F01,PEEK(&1F05)
  POKE &1F00,4
  POKE WORD &1F02,Address
  POKE WORD &1F04,Length
  MACHLG &A9,&CA,&20,&0865
  POKE &1F00,1
  MACHLG &A9,&CC,&20,&0865
 "End Bload" END FN = ERROR  :REM set implicitly
by ZBasic
 :
 LONG FN
SuperInput$(Xpos,Ypos,MaxLength,PromptChar,
DefPtr)
   :
  PromptChar = PromptChar-128 : REM clear high
bit for prompt character
  LOCATE Xpos,Ypos            : REM position
cursor
  POKE OurCH,PEEK(OurCH)-1    : REM bump counter
  :
  LONG IF PEEK (DefPtr) > 0   : REM we got a de-
fault string? (length byte > 0)
  :
  : REM Move default string to input buffer
  FOR X = 1 TO PEEK(DefPtr)
  : REM assembly module expects high bits set
    POKE InBuffer+X,PEEK(DefPtr+X) + 128
  NEXT
  POKE InBuffer,PEEK(DefPtr) : REM don't forget
to move length byte
  XELSE
    POKE InBuffer,0       : REM tell rtn no de-
fault
  END IF
  :
  REM push parms for actual input routine call
  :
  FN PushByte (PromptChar) : REM pass literal
value
  FN PushByte (MaxLength)  : REM pass literal
value
  :
  CALL InputLoc            : REM call input rtn
  :
  REM now move data from back to string
  :
  FOR X = 0 TO PEEK(InBuffer)
    POKE DefPtr+X,PEEK(InBuffer+X)-128 :REM clear
high bits for ZBasic
  NEXT
 END FN
 :
 :


 REM ----------------------
 REM Start of program
 REM ----------------------
 :
 MODE 2
 :
 FN BLOAD ("Input.Obj",1,InputLoc,0)
 PRINT@(1,2);"A better line input routine: "
 :
 TheString$ = "This is a default string."
 Prompt$ = ">"
 MyString$ = FN
SuperInput$(2,5,35,ASC(Prompt$),VARPTR(TheString$))
 :
 PRINT@(1,7);"You typed: ";TheString$
 END
```


MicroDot just $ 29.95 plus $2.50 S&H

The Logical Replacement for BASIC.SYSTEM

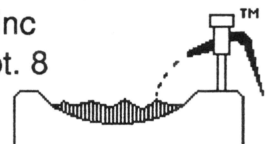


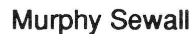
Just **2.5K** in size, but more powerful than BASIC.SYSTEM. Imagine doing BASIC overlays simply by specifying the file name and the line number where you want to overlay. How about loading an array of directory names at machine language speed. You get this and total control over ProDOS that is impossible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

Free Catalog and Details

Dealer Inquiries Invited

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





VAPORWARE by Murphy Sewall
From the May 1990 APPLE PULP,H.U.G.E. Apple Club (E. Hartford) News Letter , $15/year,P.O. Box 18027
East Hartford, CT 06118......Call the "Bit Bucket" (203) 569-8739

This is the 6th anniversary edition of this column

Can You Say October?
Although Apple officials continue to describe Macintosh System 7.0 in public as "on schedule," private sources close to Apple say the project is falling two or three days behind schedule each week and a release before harvest time is unlikely. If a version is announced this summer, it will be missing some of the anticipated features (which will become "Christmas presents"). - *InfoWorld 9 April*

New Apple II (Continued).
Apple IIgs owners saving up for a new ROM 04 machine will have to save for a new monitor too. Current Apple IIgs monitors will flicker on the new machine which will have interlaced graphics. Apple's "rethinking" of the II line's future is still said to be inconclusive, but insiders feel certain that the ROM 04 machine will be introduced - eventually. - found in my electronic mailbox

Brier to Ship 20 Mbyte Floppy.
Brier Technology has begun shipping their 20 Mbyte floptical drive (see the September and October 1988 columns) to manufacturers last month. A consumer version called the Stor/Mor will be shipped by Q/Cor (formerly Quadram) in June (about 18 months later than originally anticipated). An internal (AT bus) unit will sell for $795, an external (AT or SCSI) will be $895 and an external with MCA adapter will cost $995. Average access time is 35 milliseconds, and preformatted 3.5 inch floptical disks will be $25 each. - *InfoWorld 9 April*

The NeXT Macintosh?
Steve Jobs and John Sculley have been spotted in each other's company recently. Rumor has it that acquisition of NeXT by Apple (or perhaps the other way around?) has been on the agenda. So far Apple's board of directors have said "no" to any proposed deals. - *InfoWorld and PC Week 16 April*

Intel CPU Evolution.
Additional details about Intel's i586, i686, and i786 processors seem to be appearing quarterly (see last August, October, and January's columns). Production of the two million transistor i586 is forecast for 1992 (please allow for usual "vaporwre" slippage; that date already is a year later than predicted last July). The chip will measure 2.5 inches by 2.5 inches, and performance should be more than twice that of the i486. It will have two caches, one for instructions and one for data. The four to five million transistor i686 is scheduled for 1996, and the spectacular one inch by one inch 100 million transistor i786 is envisioned for the turn-of-the-century. According to Intel's David House, the i786 will contain four tightly coupled parallel integer processors and two vector processors operating at 250 MHz and delivering 700 MIPS. A two Mbyte cache memory will supply the six processors and the bus interface will support digital video. - *InfoWorld and PC Week 26 March*

Pen Input Systems.
Several major small computer makers and laptop vendors are on the verge of bringing out systems designed to let users substitute a pen for keyboard and mouse, but development is limited by a paucity of software. Some developers say that major applications that would let users really take advantage of pen input hardware still is a long way off. Slate Corporations, a Scottsdale, Arizona start-up, is widely acknowledged as working exclusively on applications for pen input systems. Slate has had little to say publicly about its activities - could they be the Ashton Tate or Lotus or the 90's? - - *InfoWorld 16 April*

+ Standard disclaimer applies ("The opinions expressed are my own"etc.) +

Taking a Screen Test

Jerry Kindall, Classic Apple Editor

> Saving the text screen to disk was a fairly common technique under DOS 3.3, though much of the software that did so clobbered the screen holes Jerry mentions. His program, however, writes to display memory only and is therefore a nice evolution of a useful technique. - Ross

Here's an easy way to design fancy 80-column text screens that you can display instantly in your Applesoft program. And I mean instantly: your screens pop up with machine-language speed, regardless of their complexity. No more waiting for your BASIC program to draw intricate displays, and no more painful coding of screen formats. ScreenMaker is a full-screen 80-column display editor which supports mousetext, inverse, and normal characters, and saves screens to disk in a form suitable for quick retrieval.

ScreenMaker is not a source code generator. It does not convert your screens to an Applesoft program. Instead, your screens are stored as binary files: compact, and quick to load and display.

Using ScreenMaker To Design Screens

Listing 1 is ScreenMaker, the screen editor. Save it as SCREEN. Listing 2 is the machine language code used by ScreenMaker in hexadecimal form; just enter it from the Monitor and save it as SCREEN.ML. Listing 3 is the source code for Listing 2, for hacker types.

To get going, just RUN SCREEN from Applesoft. A blank screen with a flashing cursor will appear. To move the cursor, press the arrow keys. Pressing a printable key (letters, numbers, punctuation, and symbols) puts that character on the screen at the cursor. Typing Control-M activates mousetext; while in mousetext mode, typing an uppercase letter key (and the symbols @, [,], \, ^, and _) displays a corresponding icon. Typing Control-I displays characters in reverse video. Control-N brings things back to Normal.

Control-S allows you to Save a screen. At the Save prompt, you can type ? to see a directory listing, or Return alone to cancel. Typing a valid ProDOS pathname saves the screen. Typing a 1 or a 2 saves the current screen to one of two in-memory "scratchpad" screens, which you can think of as a clipboard. They can come in handy when juggling multiple screens. Be warned, however, that they disappear when you quit ScreenMaker.

Control-L is for Loading screens. Pressing Return alone cancels; typing ? displays a catalog. You can also load from the scratchpad screens by typing 1 or 2. And, of course, typing a valid ProDOS pathname loads the screen. Typing Control-Z reverts to the last-loaded version of a screen, undoing any changes you have made since last loading or saving, including loads and saves to the scratchpad screens.

Control-A and Control-B are cursor controls. Usually, the cursor is a flashing underline. You can change it to a flashing inverse block with Control-B, and back with Control-A. Changing the cursor is only cosmetic; ScreenMaker does not have an insert mode. I included this feature mainly to allow the cursor to stand out clearly when editing different types of text.

Escape exits the program, confirming your intent first. You have to type Yes (or just Y) to get out.

Using Screens In Your Programs

You can't simply BLOAD screens created with ScreenMaker and have them appear on the 80-column screen, for two reasons. First, half of the 80-column screen isn't stored in a location easily accessible to BLOAD. The odd columns are stored beginning at location 1024 in main memory; the even columns are also stored at location

1024, but they're stored in an auxiliary bank of memory. The video circuitry displays both even and odd columns on the same screen, but you can only access half the screen (even or odd columns) at any given time.

The other, more important reason you shouldn't BLOAD screens is that the screen memory area contains undisplayed memory areas called "screenholes". There are 2048 bytes of memory in the 80-column display area, but there are only 1920 characters displayed on the screen. The rest, the screenholes, are used to store data for your peripheral cards. Changing the screenholes can cause your hardware to do strange things. So we can't simply BLOAD screens. ProDOS doesn't even begin to let you do it; if you try, you'll get a NO BUFFERS AVAILABLE message, which isn't very descriptive, but means that you're stepping on reserved memory.

"Changing the screenholes can cause your hardware to do strange things. So we can't simply BLOAD screens."

So what we need to do is BLOAD screens elsewhere in memory, then use the ScreenMaker machine language routines to display them. The machine language routines reassemble the main-memory and aux-memory screen onto the display at blinding speed, being careful not to step on the screenholes. You need a buffer (memory area) 2048 (or 2K) bytes long.

A good place to load your screens is at 2048 (hex $800). This area of memory is usually part of Applesoft's workspace, so any programs that use it will need to reserve it by running the following short program first:

```
10  POKE 103,1: POKE 104,16: POKE 4906,0
20  PRINT CHR$(4);"RUN your.program"
```

Once you've reserved that memory area, displaying your screens is as simple as this:

```
1000  PRINT CHR$(4);"BLOAD screen.name,A$800"
1010  CALL 775,8
```

The disk runs for a second or so, then the screen appears. Although it actually takes a little longer to display the screen (because of the disk access), it pops up so quickly after it's loaded that you get an impression of real speed. If you use a RAM disk there's not even any noticeable disk access.

The 8 after the CALL 775 tells the machine language routine that the screen is in memory at location $800 (it knows the last two digits are always 0), or 2048 in decimal. In technical terms, I'd say that it's the high byte of the address of a page-aligned screen buffer. The screen buffer can actually be anywhere in memory within the limits of Applesoft, but it must be page-aligned (its address must be evenly divisible by 256). By using multiple screen buffers you can have multiple screens in memory, ready to be displayed with a single CALL.

Once you've displayed your screen, you can use Applesoft's usual commands to get input, display a menu bar, or whatever. Think of your screen as a template or a background on which your program's screen I/O takes place.

Other Things You Can Do

If you'd like to save screens from some of your older programs as screen files so they can be edited by ScreenMaker, or just so they can be displayed faster, just make sure that the ScreenMaker ML routines are in memory and that a 2K buffer is available. If the buffer is at $800, as in the examples above, adding lines like the following to the program will save the screen to disk:

```
2000  CALL 768,8
2010  PRINT CHR$(4);"BSAVE screen.name,A$800,L$800
```

To use ScreenMaker's flashing cursor routine in your own program, include a line like this:

```
3000 CALL 792,223: K = PEEK (-16384) - 128: POKE -1636
```

The flashing cursor routine only waits for a key to be pressed; it does not return the key to Applesoft. The code following the CALL reads the key's ASCII code into the variable K and clears the keyboard strobe so that the next key can be read. The 223 is the screen ASCII code for a normal underscore, which will flash on the screen until a key is pressed. (Add 128 to a character's ASCII code for a normal character. Use 0-63 for inverse letters, numbers, and symbols, 64-95 for mousetext

cursors, and 96-127 for inverse lowercase.)

Try creating "text graphics" with mousetext, then saving a series of screens on a RAM disk and writing a program to display them in quick succession for animation. Or just wait for a keypress after each screen for a text slideshow.

I use ScreenMaker to "prototype" programs I'm working on. It's a lot easier to work out the bugs in a program's user interface when you can actually display how the program will look on your screen. And screens make great documentation when you want to show someone else what an unfinished program will look like. I'll include some screens from my current project on this month's disk so you can see what I'm talking about.

You may be wondering why I didn't include a fancy title screen or a help screen with ScreenMaker. C'mon, people... it's a screen editor. That means ADD YOUR OWN! In ScreenMaker, I set LOMEM to 24576 ($6000) to give me from 2048 to 16383 ($800-$3FFF) for my program and 16384 to 24575 ($4000-$5FFF) for four screen buffers. The screen buffers are for the last loaded or saved screen image (used for the Control-Z command), a temporary buffer to hold the screen when the screen must be used for something else, and the two scratchpad buffers. Since the program is relatively short, you could add additional screen buffers below $4000 if you want. Or, since the program doesn't use many variables, you could move LOMEM up a bit and add a few more screen buffers above $6000. Either way, you can add fancy "save" screens, title screens, "about" screens, help screens, and whatever else you like.

ScreenMaker could also use some more editing commands. By manipulating the text window you can add commands to scroll a section of the screen up or down, or to insert and delete lines. An easy one would be to add commands for jumping to the extreme edges of the screen (all you have to do is change the variables X and/or Y). If you're adventurous you might want to add the ability to create "text objects" which can be repositioned on the screen. Avid mouse users might want rodent control. So many possibilities, so little time.

I won't go into the assembly listing. It makes my head hurt at this time of the evening. It's pretty well commented; the only real trick I used was having a subroutine call a subroutine and then fall through into the same subroutine. It's a limited sort of recursion. The Applesoft code is pretty simple in structure. The only tricky code is in lines 210-212, which handles printing in the last position on the screen without scrolling. This is done by actually printing in position 2 on the screen, copying the screen character to its proper location, and finally restoring the character that was originally in position 2. Sneaky, but it works.

I wish you many hours of enjoyment in creating and editing your screens. May your programs gain a more professional appearance with ScreenMaker.

Listing 1: SCREEN

```
 10  REM  Screen Designer
 20  REM  by Jerry Kindall
 30  REM  ================
 40  REM  For 8/16 - Use Freely
 50  REM
 60  REM  Init Program
 70  REM
 100  PRINT  CHR$ (4);"PR#3": PRINT  CHR$
     (25);
 110  SPEED= 255: NOTRACE : TEXT : HOME :
     LOMEM: 24576
 120  PRINT  CHR$ (4);"BLOAD SCREEN.ML,
     A$300"
 125  CALL 768,64: CALL 768,80: CALL 768,88
 130 X = 0:Y = 1:KB = 49152:KC = 49168
 140 NM$ =  CHR$ (14) +  CHR$ (24)
 150 IN$ =  CHR$ (15) +  CHR$ (24)
 160 MT$ =  CHR$ (15) +  CHR$ (27)
 170  PRINT NM$;:C = 223
 175  REM
 176  REM  Main Loop
 177  REM
 190  VTAB Y: POKE 1403,X: CALL 782,C
 200 K =  PEEK (KB) - 128: POKE KC,0
 210  IF K > 31 AND Y < 24 THEN  PRINT
     CHR$ (K);:K = 21: GOTO 310
 211  IF K > 31 AND X < 79 THEN  PRINT
     CHR$ (K);:K = 21: GOTO 310
 212  IF K > 31 THEN I =  PEEK (2000): POKE
     1403,1: PRINT  CHR$ (K);:
     POKE 2039, PEEK (2000): POKE 2000,I:K
     = 21: GOTO 310
 220  IF K = 13 THEN  PRINT MT$;: GOTO 190
 230  IF K = 14 THEN  PRINT NM$;: GOTO 190
 240  IF K = 9 THEN  PRINT IN$;: GOTO 190
 250  IF K = 1 THEN C = 223: GOTO 190
 260  IF K = 2 THEN C = 32: GOTO 190
 270  IF K = 19 THEN  GOSUB 1000: GOTO 190
```

```
280  IF K = 12 THEN  GOSUB 2000: GOTO 190
285  IF K = 27 THEN  GOSUB 3000: GOTO 190
290  IF K = 26 THEN  CALL 775,64: GOTO 190
300  IF K = 8 THEN X = X - 1: IF X =  - 1
     THEN X = 79:K = 11
310  IF K = 21 THEN X = X + 1: IF X = 80
     THEN X = 0:K = 10
320  IF K = 11 THEN Y = Y - 1: IF Y = 0
     THEN Y = 24
330  IF K = 10 THEN Y = Y + 1: IF Y = 25
     THEN Y = 1
340  GOTO 190
880  REM
881  REM  Prompting for Save/Load
882  REM
900  PRINT P$;: INPUT "";A$
910  IF A$ = "?" THEN  PRINT  CHR$
     (4);"CATALOG": GOTO 900
920  RETURN
980  REM
981  REM  Save Screen
982  REM
1000  CALL 768,72: PRINT NM$;: HOME
1010 P$ = "Save screen as: ": GOSUB 900
1020  IF A$ = "" THEN  CALL 775,72: RETURN
1030  IF A$ = "1" THEN  CALL 775,72: CALL
     768,80: CALL 768,64: RETURN
1040  IF A$ = "2" THEN  CALL 775,72: CALL
     768,88: CALL 768,64: RETURN
1050  PRINT  CHR$ (4); "BSAVE"A$", A$4800,
     L$800"
1060  CALL 775,72: CALL 768,64: RETURN
1980  REM
1981  REM  Load Screen
1982  REM
2000  CALL 768,72: PRINT NM$;: HOME
2010 P$ = "Load screen: ": GOSUB 900
2020  IF A$ = "" THEN  CALL 775,72: RETURN
2030  IF A$ = "1" THEN  CALL 775,80: CALL
     768,64:X = 0:Y = 1: RETURN
2040  IF A$ = "2" THEN  CALL 775,88: CALL
     768,64:X = 0:Y = 1: RETURN
2050  PRINT  CHR$ (4);"BLOAD"A$",A$4000"
2060  CALL 775,64:X = 0:Y = 1: RETURN
2980  REM
2981  REM  Quit
2982  REM
3000  CALL 768,72: PRINT NM$;: HOME
3010  INPUT "Quit? ";A$:A$ =  LEFT$ (A$,1)
3020  IF A$ <  > "Y" AND A$ <  > "y" THEN
     CALL 775,72: RETURN
3030  POP : END
```

Listing 2: SCREEN.ML

Enter all commands exactly as printed. Be careful! You are entering important machine language code.

```
CALL-151
0300: 08 78 20 4E 03 28 60 08
0308: 78 20 80 03 28 60 08 78
0310: 20 4C E7 2C 54 C0 AD 7B
0318: 05 4A B0 03 2C 55 C0 A8
0320: B1 28 48 8A 91 28 20 3D
0328: 03 AA 68 91 28 48 8A 20
0330: 3D 03 2C 00 C0 10 ED 2C
0338: 54 C0 68 28 60 A2 00 2C
0340: 00 C0 30 09 20 4D 03 20
0348: 4D 03 CA D0 F2 60 20 4C
0350: E7 2C 55 C0 20 5D 03 2C
0358: 54 C0 A6 07 E8 20 B2 03
0360: 20 65 03 A0 F7 A2 77 B9
0368: 00 04 91 00 B9 00 05 91
0370: 02 B9 00 06 91 04 B9 00
0378: 07 91 06 88 CA 10 E8 60
0380: 20 4C E7 2C 55 C0 20 8F
0388: 03 2C 54 C0 A6 07 E8 20
0390: B2 03 20 97 03 A0 F7 A2
0398: 77 B1 00 99 00 04 B1 02
03A0: 99 00 05 B1 04 99 00 06
03A8: B1 06 99 00 07 88 CA 10
03B0: E8 60 A9 00 85 00 85 02
03B8: 85 04 85 06 86 01 E8 86
03C0: 03 E8 86 05 E8 86 07 A0
03C8: 77 60
3D0G
BSAVE SCREEN.ML,A$300,L$CA
```

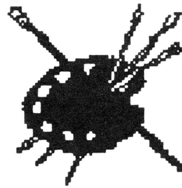
Listing 3: SCREEN.ML Source Code

```
 1  ********************************
 2  *                              *
 3  * ScreenMaker ML               *
 4  *                              *
 5  * by Jerry Kindall - 8/16      *
 6  *                              *
 7  ********************************
 8
 9           org  $300
10
11  screen0  =    $400     ;addr of screen pages
12  screen1  =    $500
13  screen2  =    $600
14  screen3  =    $700
15
16  ourch    =    1403     ;80-column HTAB pointe
```

```
17
18  ptr0     =    $00       ;ptrs to save areas
19  ptr1     =    $02
20  ptr2     =    $04
21  ptr3     =    $06
22  base     =    $28 ;base addr of current line
23
24  kbd      =    $C000
25  rdmain   =    $C054;80-column bank-switching
26  rdaux    =    $C055
27
28  combyt   =    $E74C;get comma and byte value
29
30  * CALL 768,X entry: Store A Screen
31
32  store    php            ;save interrupt status
33           sei              ;disable rupts
34           jsr  savescrn  ;actually save screen
35           plp              ;restore interrupts
36           rts
37
38  * CALL 775,X entry: Recall A Screen
39
40  recall   php            ;save interrupt status
41           sei            ;disable rupts
42           jsr  loadscrn;actually load screen
43           plp            ;restore interrupts
44           rts
45
46  * CALL 782,X entry: Flash Cursor
47
48  cursor   php
49           sei
50           jsr  combyt  ;get cursor character
51           bit  rdmain  ;point to main bank
52           lda  ourch   ;get x coord
53           lsr          ;div by 2
54           bcs  :bankok ;if odd, it's main bank
55           bit  rdaux   ;else point to aux bank
56  :bankok  tay          ;get x/2 into Y reg
57           lda  (base),y;get char under cursor
58           pha          ;save char under cursor
59           txa          ;get cursor char to acc
60  ]loop    sta  (base),y;display on screen
61           jsr  delay   ;wait a bit(or til key)
62           tax          ;save cursor char
63           pla          ;'member char from scrn
64           sta  (base),y;redisplay it
65           pha          ;save it again
66           txa          ;recall cursor char
67           jsr  delay   ;wait a bit(or til key)
68           bit  kbd     ;is key hit?
69           bpl  ]loop   ;nope, do it again
70           bit  rdmain  ;pt to main mem again
71           pla          ;pull char off stack
72           plp          ;restore 'rupt status
73           rts          ;and exit
74
75  * Wait for a while or until key is pressed
76
77  delay    ldx  #0          ;loop 256 times
78  ]loop    bit  kbd         ;is key hit?
79           bmi  :rts        ;yep, exit
80           jsr  :rts        ;kill some time
81           jsr  :rts
82           dex              ;decrement counter
83           bne  ]loop       ;not done yet
84  :rts     rts
85
86  * Save 80-column text scrn to 2K memory buffe
87
88  savescrn jsr  combyt   ;get address to store
89           bit  rdaux    ;aux bank
90           jsr  savebank
91           bit  rdmain   ;main bank, fall thru
92           ldx  ptr3+1      ;point to next 1K f
93           inx              ; save from main ba
94  savebank jsr  setptrs     ;set ptrs for move
95           jsr  saveit      ;save half this ban
96           ldy  #$F7        ;save other half
97  saveit   ldx  #$77        ;save 100 bytes
98  ]loop    lda  screen0,y   ;get from screen
99           sta  (ptr0),y    ; and store to ptrs
100          lda  screen1,y
101          sta  (ptr1),y
102          lda  screen2,y
103          sta  (ptr2),y
104          lda  screen3,y
105          sta  (ptr3),y
106          dey                  ;point to next by
107          dex                  ;decrement counte
108          bpl  ]loop           ;more
109          rts
110
111 * Load 80-column text scrn from 2K memory buff
112
113 loadscrn jsr  combyt    ;get addr to recall fr
114          bit  rdaux     ;aux bank
115          jsr  loadbank
116          bit  rdmain       ;main bank
117          ldx  ptr3+1       ;point to next 1K f
118          inx               ; load to main bank
119 loadbank jsr  setptrs
120          jsr  loadit     ;get half this bank
121          ldy  #$F7       ;get other half
122 loadit   ldx  #$77       ;get 100 bytes
123 ]loop    lda  (ptr0),y  ;get from memory buff
124          sta  screen0,y ; and store to screen
125          lda  (ptr1),y
126          sta  screen1,y
127          lda  (ptr2),y
128          sta  screen2,y
129          lda  (ptr3),y
130          sta  screen3,y
```

```
131         dey             ;point to next byte
132         dex             ;decrement counter
133         bpl  ]loop      ;more
134         rts
135
136 * Set pointers for save/load loop
137
138 setptrs lda  #$00   ;ptrs all pt to even pgs
139         sta  ptr0
140         sta  ptr1
141         sta  ptr2
142         sta  ptr3
143         stx  ptr0+1  ;point to 1st page
144         inx
145         stx  ptr1+1  ; .. 2nd
146         inx
147         stx  ptr2+1  ; .. 3rd
148         inx
149         stx  ptr3+1  ; .. 4th
150         ldy  #$77  ;init y for saveit/loadit
151         rts
```


Picture This!



Envision a full page ad for your product passing in front of thousands of the most active Apple II hardware and software buyers in the world!

And at about 10% of the cost of a similar ad in other publications!

Our ad representatives would be excited to work with you and plan an ad that would be the most cost effective for **you.**

Call (509) 923-2249 and ask for an ad kit. Or write Ariel Publishing, Box 398, Pateros, WA 98846.

"...the single most important business-oriented product for the Apple II since *AppleWorks*."

APPLE II

BY CHARLES H. GAJEWAY

Masterful database. Are you ready for a sweeping statement? Here goes: I think that *DB Master Professional* (Stone Edge Technologies: $295) is the single most important business-oriented product for the Apple II since the introduction of *AppleWorks*. As the only true relational database program for the Apple IIe, IIc, and IIGS, *DBMP* can give a 128K Apple II the kind of data-handling power and flexibility normally associated with MS-DOS and Macintosh systems running expensive and hard-to-learn software. (A relational database can link, or *relate*, information from several data files.)

I jumped right into the program with my standard test data—a pair of files that tracks a record collection, with information on album titles, artists, music category, song lengths, and composers. This test is complex, and many well-regarded programs—including *AppleWorks*—have failed miserably at it. Even with very little experience, I was able to get the system up and running with *DBMP* in a surprisingly short time.

Report generation is extremely powerful, making it easy to design anything from a mailing label, to a point-of-sale invoice (that automatically updates inventory records, of course), to customized form letters. Whereas most database programs must be combined with a word processor to do complex reports or mail merge, *DBMP* does it all.

The manuals are complete, well illustrated, and generally clear, although they are sometimes overly technical and fragmented. You will need to keep both books handy at all times, especially as you try out some of the more sophisticated features. And while the program is operated with a simple menu system, *DBMP* takes a fair amount of time to learn because of its array of features and options. *DBMP* gives you all the power you need and can even import your current files from *AppleWorks* (except version 3.0) and other programs. ■

Reprinted with permission from *Home Office Computing*.

DB Master *Professional*

Stone Edge Technologies, Inc.
P.O. Box 3200 • Maple Glen, PA 19002 • (215) 641-1825

Directing Traffic

By Nate Trost

(Editor: Nate might not appreciate me sharing this, but were it not for a slight over abundance of exclamation points it would be downright difficult to discern that he is but 13 years old.)

No, this isn't an article about using an Apple II to control McDonald's, but it *is* about how you can easily create multiple stacks and direct pages!

Directing Direct Page

OK, let's say that you're writing a program that requires a lot of direct page space. Since DP space on the GS is at a premium, if you need more, just create your own! Yes, it's easy, it's simple, and it works!

You can do this by using a register that premiered on the 65816, the Direct Page Register (DPR). Unlike zero page on the 6502, which is fixed in memory from $00 to $FF, the direct page can be anywhere in bank zero. The DPR is 16 bits, so it can reference a value anywhere within the 64K of bank zero. When you do an operation that accesses direct page, such as STA $20, the 65816 uses the address in the DPR to determine the first byte of the direct page (example: if the DPR is set to $3300, then a LDA $12 will be reference location $3312). Listing one is a short program that demonstrates this technique. It creates space for a new DP, stores $1234 in $20 of the normal DP, switches to our new DP and stores $ABCD in $20, then checks to see if both DPs are correct.

"BUT WAIT!" you cry, "you can't just put your direct page anywhere you want on the GS, what about the memory manager? It has to be relocatable!" Yes, you are right! So, we use _NewHandle to get a page of memory from the Memory Manager, turn our handle into a pointer and pass that to the DPR. Here's the code:

```
PushLong  #0        ;space for result
PushLong  #$100     ;one page of memory
PushWord  ProgID    ;our program ID
PushWord  #$C005    ;more on this later
PushLong  #0        ;we want bank 0!
_NewHandle    ;old-fashioned _ tool-call
PullLong  NewDPH   ;get handle
DeRef NewDPH;NewDPPtr;deref hndl to get
* a pointer to our block of memory.
```

The only real head-scratcher is the PushWord #$C005. This tells the memory manager that we want our new memory block locked (unpurgeable), fixed (unmovable), in a fixed bank, and page aligned (starting on a page boundary).

"Fine, but...how do we get the pointer into and out of the DPR??" Glad you asked. There are two opcodes for doing this: TDC (transfer DPR to accumulator) and TCD (transfer accumulator to DPR). Here's how the code looks to make our new direct page active:

```
tdc          ;pop old DP loc. into acc
sta OldDP    ;save it away for later

lda NewDPPtr ;load our new location
tcd          ;and WHAMMO!
```

To switch back, just STA NewDPPtr and LDA OldDP. So there you have it—an easy, efficient way to get the direct page space you need. Remember to get rid of your new DP space via _DisposeHandle when you're done with it!

Stocking Stacks

Creating another stack is quite similar to the DP code above, with a few minor differences.

1) Rather than the direct page register, we use the 16-bit register called the Stack Pointer (SP), which points to the location that the data will be stored next time you push something on the stack.

2) The _NewHandle call is the same, but the stack is not limited to one page! You can have a 5-10K stack if there is enough room in bank zero.

3) After you dereference your memory handle, you must

add the size of the handle minus one (so it points to the HIGHEST byte in the allocated block, e.g. a one-page block of memory for a stack at $100 means that the SP should be set to $1FF, $200 to $2FF, and so on). This is because, when you push a value to the stack, the SP is decremented. If we set the stack pointer to the beginning of the memory block, we would push our way right out of our memory.

4) The opcodes are slightly different: TSC and TCS transfer the SP to and from the accumulator. Here's the code:

```
tsc          ;get old SP in accumulator
sta OldStackPtr ;and save for later

lda NewStkPtr   ;load new pointer
tcs             ;and pop it in
```

Since the new stack pointer will change as you push/pull to switch back to the old stack, do this:

```
tsc             ;get our SP in accumulator
sta OurStk      ;and save for later

lda OldStackPtr ;get old stack
tcs             ;and make the switch
```

Then, you may use OurStk instead of NewStkPtr for your next stack switch operation. Listing two demonstrates this method by pushing stuff on both stacks, making sure the values are correct, doing a JSR and a tool call using the new stack, and returning to the calling routine.

These two tricks can be used in any GS program. You may even want to turn the stack and direct page switching code into a macro (e.g. ~StackSwitch OldStack;NewStack)! The only things you have to really watch are:

1) That you don't put values in one stack or DP and try to get them out of another, and

2) Pushing or pulling too much data on or off the stack.

Listing 1

```
        lst   off
        xc
        xc
        mx    %00
        rel
        use   SDP.MACS
        put   1/tool.equates/e16.gsos
        put   1/tool.equates/e16.memory
**********************************************
*                                            *
*  === Double DP === V 1.0    By Nate Trost  *
*--------------------------*
*  Creates second DP and fools around with   *
*  it, switch DP's a couple times.           *
*                                            *
*  Copyright (c) 1990 Ariel Publishing and   *
*  Nate Trost. Some rights reserved.         *
*                                            *
**********************************************

*-----------------------
*--- first let us start our tools---
*-----------------------

        _TLStartUp        ;the all-wise TL

        ~MMStartUp #0     ;Mr. Scrooge ze MM
        PullWord ProgID   ;grab our program ID

        _MTStartUp        ;start Misc. tools

*-----------------------
*-- allocate our memory from Mem. Manager --
*-----------------------

* Get 1 page of locked,fixed,bank 0,page aligned

        PushLong #0       ;space for result
        PushLong #$100    ;only 1 page
        PushWord ProgID   ;program ID
        PushWord #$C005   ;memory attributes
        PushLong #0       ;we want bank 0!
        _NewHandle        ;I'm hungry for RAM!

* Now we gotta get handle and turn into pointer

        PullLong DP2Hndl
        Deref DP2Hndl;NewDPPtr

*-- save old DP contents of $20
*-- and put our own stuff on

        lda   $20
        sta   Old20
        lda   #$1234
        sta   $20

*-----------------------
*--- OK! now switch to 2nd DP & test it by --
*-- saving some stuff in it and then..... --
```

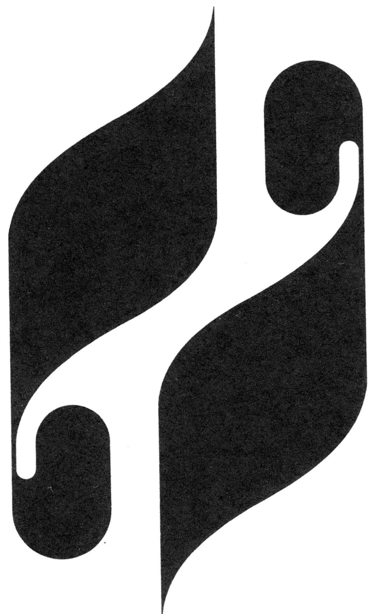
```
*----------------------------

        tdc                 ;get old DP ptr.
        sta    OldDP        ;and save it

        lda    NewDPPtr     ;get ptr to our space
        tcd                 ;turn it into new DP

        lda    #$ABCD       ;easy to remember!
        sta    $20          ;PRESTO! even though
                            ;$0 in direct page
                            ;isn't $0 in real
                            ;memory, IT WORKS!

*-------------------
*--- switch to old DP ---
*-------------------

        tdc                 ;save current DP Ptr.
        sta    TwoDP

        lda    OldDP        ;switch to old DP
        tcd

Whoops  lda    $20          ;load the value we
        cmp    #$1234       ;stored earlier
        bne    Whoops       ;is it the same?
        lda    Old20        ;yes, it was
        sta    $20          ;so restore old value
*---------------------
*--- now back to the new DP ----
*---------------------

        tdc
        sta    OldDP        ;save old DP Ptr.

        lda    TwoDP
        tcd

TryAgain lda   $20          ;load value stored in
        cmp    #$ABCD       ;new DP & check if
        bne    TryAgain     ;it's correct

*---------------------
*--- back to the old DP ---
*---------------------

* We don't save current DP ptr cuz we won't
* be using it anymore....

ByeBye  lda    OldDP
        tcd

*--------------------------
*--- Dispose our DP space, shutdown ---
*--- tools & get outta this place!  ---
*--------------------------

        PushLong DP2Hndl ;clean up memory
        _DisposeHandle

        _MTShutDown      ;shut down MT
        ~MMShutDown ProgID
        _TLShutDown      ;and MM & TL & BLT...

        iGSOS _Quit;:QParms;1
:QParms ds     2
        ds     4

        brk              ;should NEVER hit


*- >> Data for program <<-*

OldDP    ds    2
DP2Hndl  ds    4
ProgID   ds    2
NewDPPtr ds    4
TwoDP    ds    2
Old20    ds    2

*==============================================

        sav    SDP.1
        end
```

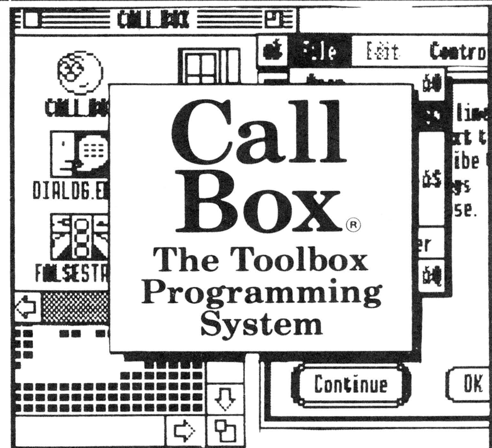
Listing two:

```
        lst    off
        xc
        xc
        mx     %00
        rel
        use    SS.MACS
        put    1/tool.equates/e16.gsos
        put    1/tool.equates/e16.memory
************************************************
*                                              *
*  === Double Stack === V 1.0  -By Nate Trost *
*------------------------*
*  Creates second stack and fools around with  *
*  it, switch between stacks a couple times.   *
*                                              *
*  Copyright 1990 Nate Trost and Ariel         *
*  Publishing Inc., but go ahead and use this  *
*  code in your own program.                   *
*                                              *
************************************************

*----------------------
*--- first let us start our tools---
*----------------------
```

```
        _TLStartUp          ;the all-wise Tool Loc.

         ~MMStartUp #0      ;Mr. Scrooge ze M.M.
         PullWord ProgID    ;grab our program ID

         _MTStartUp         ;start Misc. tools

*————————————————————————
*—— allocate our memory from Mem. Manager ——
*————————————————————————

* Get one page of locked,fixed,bank 0,
* page aligned memory

         PushLong #0        ;space for result
         PushLong #$100     ;only 1 page of memory
         PushWord ProgID    ;program ID
         PushWord #$C005    ;the code for attribs.
         PushLong #0        ;we want bank 0!
         _NewHandle


*Now I get memory handle and turn into a Pointer

         PullLong Stack2H

         Deref Stack2H;NewStkPtr

*Now start the testing.....

         lda    #$b500      ;push value onto old
         pha                ;stack to test later

*————————————————————————
*-OK! now switch to 2nd stack & push stuff on-
*————————————————————————

         tsc                ;get old Stack ptr.
         sta    OldStack    ;and save it

*Now we get our pointer and make it point to
*the LAST byte of our memory because when you
*push the Stack Pointer is DECREMENTED

         lda    NewStkPtr   ;get ptr
         adc    #$ff        ;point to LAST byte
         tcs                ;and make new stack

         lda    #$3F        ;push some data on
         pha
         lda    #$542C
         pha

*————————————————
*——— switch to old stack ———
*————————————————

         tsc                ;save stack Ptr.
         sta    TwoStack

         lda    OldStack    ;switch to old stk.
         tcs

Mistake  pla                ;test our old stack
         cmp    #$b500
         bne    Mistake

*————————————————————
*——— now back to the new stack ———
*————————————————————


         tsc
         sta    OldStack    ;save old S. Ptr.

         lda    TwoStack
         tcs
         pla                ;test the stuff put
         cmp    #$542C      ;on the new stack
         beq    Next
         lda    #4
         brk                ;should NEVER BRK
Next     pla
         cmp    #$3F
         beq    TestJSR
         lda    #3
         brk                ;should NEVER hit

*————————————————————
*——— Now let's jump to a subroutine ———
*——— and make a tool call using new ———
*——— stack........................ ———
*————————————————————

TestJSR  jsr    SubR

*——————————————————
*——— back to the old stack ———
*——————————————————

* Since we won't be using stack2 anymore, we
* don't save the current stack pointer.

ByeBye   lda    OldStack
         tcs

*————————————————————
*—— Dispose our stack space, shutdown ——
*—— tools & blow this joint.......... ——
*————————————————————

         PushLong Stack2H   ;clean up our mess
         _DisposeHandle

         _MTShutDown        ;shut down MT
```

```
        ~MMShutDown ProgID
        _TLShutDown       ;and MM & Tool Loc.

        iGSOS _Quit;:QParms;1
:QParms ds    2
        ds    4

        brk               ;should NEVER hit

*—————————————————
*—— Our test subroutine ——
*—————————————————

SubR    ~NewHandle #175;ProgID;#attrLocked;#0
        _DisposeHandle    ;testing tools calls
        rts               ;with new stack..


*— >> Data for program <<—*

OldStack ds    2
Stack2H  ds    4
ProgID   ds    2
NewStkPtr ds   4
TwoStack ds    2

*============================================

        sav   SS.1
        end
```





WYSIWYG?

(What You See Is What You Get)

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

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

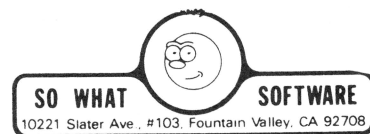
All editors output APW source code, Linkable object code or resource files to make the best match to your current development system. Everything is accessable from the CALL-BOX Editor shell that includes these editors plus File utilities, Configuration utilities, programmable application launcher and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

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

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

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

(714) 964-4298



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.

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 "Illusion's of Motion".

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.

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

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

- FRAME.FN
- 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).

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

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

Insecticide

The typesetting gremlins got loose last month, wreaking all kinds of havoc in Robert Stong's parameter passing article for Applesoft.

• First, change line 100 of the Applesoft listing to:

```
PRINT CHR$(4);"BLOAD PARAMS.OBJ, A$6000"
```

• Next, beware that our right margin was extended too wide, thereby causing the last character or two of several lines to get cut off. Line 520 needs a ":", Line 540 needs TH changed to THEN, and line 615 lost the final ")".

Also, the version of the program we printed had already been run, thereby causing the variables names in the subroutine to have been changed! This does not hurt the program, but it makes the REM statements incorrect.

Thanks, Bob, for being so prompt in catching these things for us.

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

The Sensational Lasers

Apple IIe/IIc Compatible

SALE $345 *Includes 10 free software programs!*

New! Now Includes COPY II PLUS®

The Laser 128® features full Apple® II compatibility with an internal disk drive, serial, parallel, modem, and mouse ports. When you're ready to expand your system, there's an external drive port and expansion slot. The Laser 128 even includes 10 free software programs! Take advantage of this exceptional value today............ ***$345***

Super High Speed Option!

only **$385**

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

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

DISK DRIVES

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

Save Money by Buying a Complete Package!

THE STAR a LASER 128 Computer with 12" Monochrome Monitor and the LASER 145E Printer $620.00

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

ACCESSORIES

* 12" Monochrome Monitor .... $ 89.00
* 14" RGB Color Monitor $249.00
* LASER 190E Printer $219.00
* LASER 145E Printer .... New! $189.00
* Mouse .... $ 59.00
* Joystick (3) Button .... $ 29.00
* 1200/2400 Baud Modem Auto $129.00

USA MICRO YOUR DIRECT SOURCE FOR APPLE AND IBM COMPATIBLE COMPUTERS

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

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

http://apple2scans.net