/// Cheers!

Volume 1 Number 1 | Summer 1984 | A.P.P.L.E.

Apple

The Apple /// Magazine
Premier Issue

VINTAGEMICROS

COLLECTIBLE HARDWARE,
SOFTWARE, AND MEMORABILIA

www.vintagemicros.com

***The best choice
for all your vintage
Lisa and Apple hardware
and software needs***

Apple Pugetsound Program library Exchange

CLUB INFORMATION

General Offices

A.P.P.L.E. Headquarters
7 Jogues Road
Winnipeg, Manitoba
Canada
R2J 1Z6

A.P.P.L.E. Publishing Office
28-9-2810 Motomachi
Tokorozawa Japan 359-1121

Club Officers

President Bill Martens
Vice Pres. Rick Sutcliffe
Secretary Masako Nozaki

A.P.P.L.E. Board of Directors
Chairman Val J. Golding
Director Bill Martens
Director Rick Sutcliffe
Director Mike Pfaiffer
Director Chris Hill

Advisor Richard Hubert
Advisor Mike Thyng

Consultants
Apple III / Lisa: David Craig
Apple II SW: Bill Martens
Apple II HW: John Woodall
Mac OS 7-9: Bryan Villados
OS X / iMac: Rick Sutcliffe
Emulation: Bill Martens

Call-A.P.P.L.E. Staff
Managing Editor
Val J. Golding

Production Editor
Bill Martens
Mike Pfaiffer

Research Manager
Bryan Villados

BBS Sysop
Todd Nathan

A.P.P.L.E. Doctor
Robert Platt

Archivist
David Craig

Editorial Department

To Contact the Call-A.P.P.L.E. editorial department, you can send email to editor@callapple.org

Advertising

Magazine Advertising
ads@callapple.org

Call-A.P.P.L.E. Hot Line

The Call-A.P.P.L.E. Hotline support is available 24 hours per day by email via support@callapple.org

Bulletin Board

A.P.P.L.E. Crate
www.callapple.org/members/bbs
sysop@callapple.org

Membership Information

Annual Membership $25.00 USD

Membership is on an annual basis commencing with the month of enrollment. All memberships are sold on-line only through our web page at www.callapple.org/join

Web Services

Editor: editor@callapple.org
Sales: sales@callapple.org
Info: info@callapple.org

Unclassified Ads

This system is available to all of our readers and members with no charge for listing items.

www.callapple.org/classifieds

Memberships and Orders
Memberships are automatically approved online when you join through the A.P.P.L.E. web site and your member ID is issued at that time. Please allow 10 days turn around on all orders for items from the A.P.P.L.E. catalogs that are purchased offline. All items purchased online which are downloadable will be received immediately. For our online catalog, go to www.callapple.org/cat/

Writing for Call-A.P.P.L.E.
Call-A.P.P.L.E. is an all-volunteer magazine and thus we encourage members of the Apple community to submit programs and reviews to our editorial department for publishing.

If you have a program you would like to share with the Apple community through our magazine, send all documentation and source code to submissions@callapple.org. Please be sure that your documentation and source is correct prior to submitting the program or review.

Copyright
The contents of this magazine are copyright Apple Pugetsound Program Library Exchange. All Rights Reserved. Articles may be re-printed by Apple User Groups as long as all credits are left intact.

Apple is a registered trademark of Apple Computer Inc. All other copyrights are the property of the respective owners.

ISSN 1705-4117

Club Meetings
The A.P.P.L.E. Users group meetings are held online each month on the 1st Thursday evening of each month from 6pm to 8pm PST. This meeting is open to all members and non-members alike. The meetings will occur online and can be accessed through the A.P.P.L.E. Website. (www.callapple.org)

RainyDay Internet

Table Of Contents

Editorial

Threesies: New Products and News

Three Ideas: Handy Tools For Business BASIC

SetDebug: You CAN get back from the Monitor!

A Simple List Manager In Pascal

Business BASIC: A Utility from 'the Lazy H'

Review: A Closer Look At VersaForm

Review: Impressions of /// E-Z Pieces

Author Guidelines

Print Articles On Paper

EDITORIAL: /// Cheers — THE FIVE "W"s

Dave Lingwood

In Journalism classes long ago I had drilled into me the five "W"s of reporting: "who, what, where, when, and why?" Since my purpose here is to describe /// Cheers, I figured what the heck ... Here goes:

WHO?

/// Cheers is brought to you by A.P.P.L.E., Apple PugetSound Program Library Exchange, which is, as you no doubt know, the world's largest Apple users' roup. We grew beyond the functions of a local club years ago — chiefly by being there first — into what amounts to a combined hardware/ software co-op and information service organization for our over 20K members. We also publish "Call-A.P.P.L.E." magazine each month. More specifically, /// Cheers is a service of the /// Special Interest Group of A.P.P.L.E. One of several such groups formed in the last year, // /SIG is a collection of folks with shared interests and needs for help.

WHAT?

/// Cheers is a disk-based publication, sold to all comers (NOT just A.P.P.L.E. members). Why do it on disk? Well, publication costs, timeliness, and usefulness are the three main arguments. A disk magazine doesn't have production costs: you just duplicate as many copies as you need whenever you need them (the ultimate copying machine??). We can also put articles and programs together more quickly: authors are required to submit machine-readable copy and programs. Programs are more rapidly useful if you don't have to keystroke the silly things in — and then go bug hunting.

Each /// Cheers comes with text and programs in both Business BASIC and Pascal. The "CHEERS" program which boots on the first diskette is the operating heart of /// Cheers. It shows you the table of contents, the text of articles and Pascal programs.

In terms of content, /// Cheers's editorial policy is still evolving. The general goal is to be as helpful as possible to /// users. In the dark of the night, wrapped up in one's own project, it often is quite a lonely thing to be a /// user. // / Cheers wants to help end that, building a core of information and techniques on one hand, and a network of people on the other — the same kind of information and linkages that enabled the Apple][software/hardware community to grow so quickly.

WHERE?

A.P.P.L.E. is based in the Seattle area. Our mailing address is:

A.P.P.L.E.
21246 68th Ave. S.
Kent, WA 98032

The office phone number is (206) 872-2245. Hours are 9AM-4PM (Pacific). Mike Christensen is the staff member responsible for /// Cheers.

Our hot line for members' technical questions is (206) 872-9004. Hours are 9AM-3PM and 6PM-10PM, daily. A list of /// consultants is carried every month in Call-A.P.P.L.E. Orders (only): (800) 426-3667, 24 hours.

You can get /// Cheers, either individually or by subscription. Individual copies are for sale through A.P.P.L.E., and from dealers (if your dealer doesn't have it, scream — once again, no doubt — about his level of support for the ///, and demand that he stock it). Subscriptions are available to A.P.P.L.E. members.

Single-issue price is $12.50. Subscription (A.P.P.L.E. members): $40.00 per year (4 issues).

WHEN?

As we have said, quarterly. At least to start. It is frankly a supply and demand issue: if you folks want more, and authors can supply us to meet that need, then we'll expand.

WHY?

Now the fun part: what amounts to an editorial on the /// and needs of people using it.

We all know the initial problems with the /// hardware. The bad taste left by the early bugs still lingers — even though Apple Computer quietly did the unheard of by replacing all of those early, buggy systems.

The bigger problem was what I consider to be a classic blunder on Apple Computer's part in the timing of release for developmental software. If you remember back, there was a long period there when all that was available for the /// was Business BASIC and an information void tied to the rather "locked" nature of the machine. Face it, commercial developers don't do much in BASIC anymore. In effect, the Pascal elitism that dictated that an Assembler not appear until Pascal did doomed the /// to second-class status. By the time the development tools did appear, the IBM was on the horizon, and developers were already aiming their big guns at this lucrative market.

The /// is also a higher priced machine than the][. Software for expensive computers is developed by those who can afford the equipment. Individual buyers still bought (and still do buy) the Apple][. Companies who develop software do so with a bottom-line orientation for the size of the market and the cost of gathering the information needed to develop. Yet, the /// lacked the unpaid knowledge building corps from which the][software development industry benefited: the thousands of (originally) hobbyists and professionals who dug out the technical facts. Apple Computer didn't fill the void with details about the "innards" of the ///. The result? The knowledge vacuum and smaller market sucked much of the vitality out of the ///.

Two years ago Apple began the "Third Wave Developer" program, providing equipment and technical support for developers to overcome the software gap for the ///. Their /// support group was the most responsive and helpful I've ever encountered within Apple. They began to pull the network of producers and users together. I only wish it had all happened two years earlier.

ProDOS for the][, was in part designed to help the ///, by providing compatible operating systems for the][and ///. It makes the /// user/developer feel somewhat like the old craftsman whose neglected knowledge becomes once again a valuable commodity when fad interest in his/her topic hits the society: pleased, but with a bitter tinge. ProDOS will also force part-time /// developers to dig deeper into SOS, since the compatibility is at the SOS call level. Too little, however, and much too late — my feeling is that ProDOS only complicates life for the][user, without really helping the ///.

Things have changed for the worse since then, and recent developments at Apple have us all worried about the future of the ///. Apple Corporate decided that the /// wasn't getting enough market share, and (as of Spring '84) stopped internal development work on the /// — placing the support issue back in all of our collective laps again. Now we're all waiting for the other shoe to drop: the expected announcement that /// production itself will be stopped. It won't surprise us when it comes.

What a sorry state of affairs for a machine that still holds its own, when armed with good software, against any PC you care to name.

Bitching won't help. With Apple bowing out, we /// users will need all the information help and network building support we can get — and it looks as if we'll have to do it ourselves. We need a larger tool box. We need to probe further into SOS for development work. We need better knowledge of new products (both applications and tools). /// Cheers will try to help in both areas. Just try to remember that adversity breeds creativity.

Our first issue is geared pretty much toward tools, particularly utilities. Remember: first you tool up the factory, then you go into production.

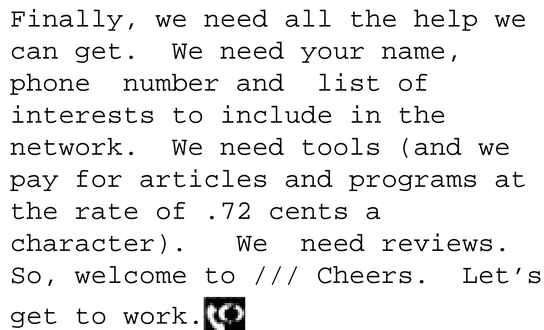
Soon we will begin the network-building job. In an upcoming issue we'll include a simple database program, with data on A.P.P.L.E. members interested in the ///: names, phone numbers, areas of application. Over time, new names to be added to that database will be included.

As we get the tools under control, we will expand more toward applications with reviews and lists of new products. We will, however, guard the unique nature of a disk based publication. Machine readability is more helpful for programs than for text. As disk space becomes a premium, we'll tilt more in favor of programs, particularly if adequate reviews exist elsewhere. Not being particularly proprietary, we'll simply announce what is available elsewhere.

In fact, another useful feature may be a database on /// coverage

in other magazines. What do YOU think?

Finally, we need all the help we can get. We need your name, phone number and list of interests to include in the network. We need tools (and we pay for articles and programs at the rate of .72 cents a character). We need reviews. So, welcome to /// Cheers. Let's get to work.



THREESIES:

New Products and News

WHAT AN APPLE /// CAN DO?

Wave this Apple Computer publication under the nose of those who ask. For $3.00 you get 96 pages of product descriptions, software supplier names and addresses, and a list of software by application category. Available from your dealer or Apple Computer. Make sure you get the latest issue, released in April, which added more software and hardware.

MOUSE, MICE, MEECE

If you haven't heard, Apple Computer has announced a mouse and driver board for the //, priced at $150. The same product will work on the /// as well. All you need is the /// mouse driver software, which Apple Computer will be supplying.

/// PLUS

It is HERE. Features of this update include a //e-style keyboard (finally a DEL key!), interlaced video for double text/graphics screen resoultio|, clock/calendar function, SOS 1.3, and a redesigned mother board with a larger power supply.

List price on the /// Plus is $2995. An Interlace Upgrade Kit is available for earlier ///s, and the Clock/Calendar Kit is $35 (plus installation).

NEW TECHNICAL PRODUCTS FROM APPLE FOR DEVELOPERS

These should be available now from dealers, or certainly from Apple. They are packaged and priced separately.

Apple /// Pascal Tool Kit:
Development help for Pascal folks. It includes utilities for such programming functions as compiling, comparing data text files, designing user interfaces, and directory sorting.

Apple Pascal Numerics:
For both the // and ///. Units to give Pascal programmers single, double and extended-precision real and integer numbers. Incorporates IEEE-standard numerics and math functions.

Pronto: The Apple /// Pascal Debugger: Debug while executing programs at full speed, without recompiling.

OTHER PUBLICATIONS

On Three: Box 3825 Ventura, CA 93006. Monthly, $30/year. Programs, reviews, etc.

Open Apple Gazette: Original Apple ///rs Box 813 San Francisco, CA 94101
A users group newsletter. Programs, reviews, lists of /// software.

TECHNICAL MANUALS

There are now four such available from Apple Computer:

1. SOS Reference Manuals, V 1&2, (includes ExerSOS Disk), A3L0027, $50.
2. Apple /// Pascal Technical Reference Manual, A3L0006, $50.
3. Device Driver Writer's Guide, A3L0023, $25.
4. Apple /// Technical Reference Manual (not available from dealers).

APPLE SERVE ///

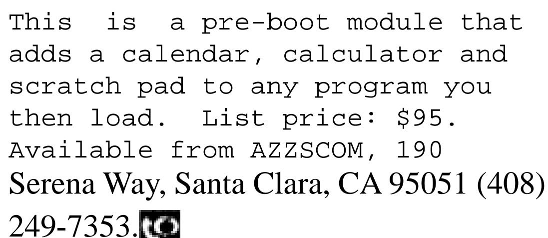
Apple Computer has created this on-line information service for /// owners, developers and dealers. It is available on CompuServe, and you'll need your own CompuServe account (for which you'll pay, of course).

It provides a bulletin board, "electronic newsletter" from Apple on new products and applications, info on software updates, etc. You can also set up electronic mail connections to other /// users.

For more information contact:

Mr. Albert Chu
Apple Computer
Mail Stop 22-A
20525 Mariani Ave.
Cupertino, CA 95014

CALENDAR PAK


This is a pre-boot module that adds a calendar, calculator and scratch pad to any program you then load. List price: $95. Available from AZZSCOM, 190 Serena Way, Santa Clara, CA 95051 (408) 249-7353.

THREE IDEAS:
Handy Tools for Business BASIC

Text by Dave. Lingwood
Programs by Brian Matthews
Action-Research NW

BASIC COMPARISONS

Most Apple /// Business BASIC users also work with Applesoft, either from earlier][days, or through emulation mode. Business BASIC (hereafter "BB") has all the professional features you need, but it lacks the flexibility provided by the openness of the][. This article and attached programs recount some of the pitfalls we encountered and useful tricks devised in transferring a large statistical analysis package from Applesoft to BB. The tricks add back some of the flexibility of Applesoft.

1: ARRAY HANDLING AND REQUEST.INV

If you haven't already discovered it, REQUEST.INV, provided with BB, contains very useful routines for rapid reading and saving of numeric data arrays. They are FILREAD and FILWRITE.

FILREAD/FILWRITE do a read or save directly into/from a vector. The data are saved as a binary file on the disk. The PERFORM command tells the invokable the name of the array or vector, the file name, and the number of bytes to transfer.

Let's assume you have an array DIMensioned as DD%(N) which contains data you want to save

save to disk, then re-fill with other data on the disk. Here are the appropriate BB statements to do it:

```
   10 INVOKE"request.inv":REM at
beginning of program
   20 DIM RW$(1): RW$(0)= "Read":
RW$(1)= "Writ"

   100 filename$= "dataout": RW=
1: REM write data
   110 GOSUB 1000
   200 filename$= "datain" RW= 0:
REM read data
   210 GOSUB 1000

   1000 REM Subroutine for array
read/write. RW=0 to read, =1 to
write
   1010 PRINT RW$(RW); "ing array
DD%"
   1020 nbytes%= (N+1) * 2:REM 2
since integer array
   1030 datafile%= filenumb: REM #
of data file
   1040 DD$= "DD%": REM name of
array to transfer
   1050 OPEN#datafile%,filename$
   1060 IF RW THEN PERFORM
FILWRITE(%datafile%,@DD$,%nbytes%)
   1070 IF NOT RW THEN PERFORM
FILREAD(%datafile%,@DD$,%nbytes%,@count%):
        IF nbytes%<>count% THEN
PRINT "Err: Read file <> array
length!"
   1080 CLOSE#datafile%
   1090 RETURN
```

The RW variable controls reading or writing. DD$ is the important value passed to FILREAD or FILWRITE which tells it the name of the array to find in memory, and from the beginning (0th cell) of which to begin reading or writing "nbytes%" bytes. The "count%" variable is returned after a read, containing the number of bytes actually read. The IF test after the read checks that this figure agrees with the number of bytes in the array (though you might want to read fewer bytes into an array, you should never read more!).

You could even put the names of various arrays into a string array, then assign DD$= ARRAY$(k), for example, or pass the array name into the subroutine as a parameter. There is a lot of flexibility here, and more importantly, a lot of SPEED. Data transfer is much faster than with INPUT/PRINT.

2: DIMENSIONS, STRINGS AND THE 256K ///

A problem we bumped into almost immediately with the stat package was the limit on array size, and the rule about string arrays in BB that affects the 256K machine. Simply put:

a. No numeric array can be longer than 64K
b. Strings must be defined in the first 64K block of program memory.

The first, while a pain, you can live with by careful program design (especially with FILREAD/ FILWRITE available). The second causes real speed problems if you have many string arrays. Being an interpreter, BB has to scan through the list of all arrays to find the one you want. Normally you'd define your fast numeric arrays first; but if your numeric array is a full 64K in size, that would force any string arrays subsequently defined to cross into the second 64K block, generating an error that is not mentioned anywhere handy:

```
Error code 21!
Variable/memory error
```

Again, it means you've accessed a string that crosses the block

boundary on a 256K machine. It brings the program to a screeching halt!

Once you know this, the cure is to dimension all the strings first. Uhg. Speed tumbles as the program wades through all the string arrays to find the numerics. The solution we came up with, though a bother, was to dimension just one string array, then store all the various command lists, etc., in it, using an offset variable to find the "base" of each list.

For example, if we had 10 commands, 6 options, and 20 error codes the master string array would look like this:

```
     Cmdlist=0  strings cmdlist+1—
cmdlist+10 are commands
     Optlist=10    "    optlist+1—
optlist+6   "  options
     Errlist=16    "    errlist+1—
errlist+20  "  error messages
```

What a classic pain. But, it works.

3: BLOAD+BSAVE+PEEK+POKE = PROGRAM OVERLAY

A heck of an equation. Why resurrect these commands we thought the superior design of the /// made unnecessary? The answer lies in the heart of our program design, and is related to disk space and speed.

The AIDA statistical package consists of a chunk of common code that is always used to process commands, read and save data, etc., plus specific code that is used depending on which command the user gives. The traditional way to handle this is to have a main "menu" program chain various separate programs depending on which command is given. The trouble with this is that the common code has to exist in each chained program. That means longer read time and disk space wasted through duplication of that code.

The][version already had solved this by inventing program overlay (see "Call-A.P.P.L.E.," Nov., 1980). The various command-specific program segments were saved as binary files, then BLOADed quickly right into the middle of the running program, which never knew what had hit it. When we converted to the ///, the improved CHAIN was faster than would have been the case on the][, but there just wasn't enough disk space for the program!

Back to the drawing board. The way overlay worked on the][was to find the RAM address of one line of the program: a simple subroutine peeked the Applesoft zero page locations containing the current line pointer — in effect the one-line subroutine found its own address in memory. Then the code was BLOADed, the only trick being that each "module" so loaded must be shorter than the original "back end" of the program, so as not to overwrite the data.

Brian began to dig around in memory, and added a lot to what we had previously known about reserved locations in BB. Once he found the statement pointer he was able to create the needed "overlay" statements by peeking the counter, then BLOADing the module to that point — after creating PEEK, POKE, BLOAD, and BSAVE, of course!

First, let's describe those four new commands, which are useful in

their own right. Later we'll combine them into the form needed for program overlay.

Peeking and Poking

Peek and Poke are defined in the AIDA.TOOLS invokable. Both routines are set up to peek or poke one- or two-byte values, and to use the "extend byte" for the RAM banks. The syntax is:

```
   x =
exfn%.peek(%address,%xtend,%length)
   PERFORM
poke(%address,%xtend,@VALUE%,%length)

   where:  address = address of
low byte (0-65535)
           xtend   = extend byte
(0-3)
           length  = 0 for one-
byte, 1 for 2-byte
           VALUE%  = value to poke
```

For example:

```
   VALUE%= exfn%.peek(%61,%0,%1)
   VALUE%= VALUE% + 512: PERFORM
poke(%61,%0,@VALUE%,%1)
```

The peek will set variable "VALUE%" equal to the two-byte data found at locations 61 (low byte) and 62 (high byte) decimal in the "true" (xtend=0) zero page. The poke will poke the two-byte value contained in "VALUE%" into 61 and 62. This would add 512 bytes onto the BB end of program pointer (clobbering variables, by the way). Careful: poking in particular assumes that you know WHAT you're poking, and why. The list of zero-page and other BB locations found elsewhere on this disk provides valuable guidance.

Bload and Bsave

The syntax for these commands, set up as invokables, looks like this:

```
   PERFORM
bsave(@filename$,%address,%xtend,@savelength%)
   PERFORM
bload(@filename$,%address,%xtend,@foundlength%)
```

Before BSAVE the number of bytes to be written must be stored in the "savelentth%" variable. After a BLOAD this same variable will contain the number of bytes actually read from the disk.

All four of these commands are contained in the AIDA.TOOLS invokable described at the end of this article.

Overlay

This technique lets you break long programs into shorter pieces, where you don't want to have to re-read (as with chain) extensive common code. The program must be laid out like this:

```
   Common code (low line numbers)
   Transition subroutine (finds
its own RAM address)
   Overlay "module" (high line
numbers)
```

When the program is first loaded and run, initialization or startup code is contained in the overlay area. This initialization code MUST be longer than any later module will be, even if you must pad it out with long REM statements. Later this initialization code will be overwritten with the binary modules successively BLOADed.

There are a few other rules for writing the initialization code, based on the obvious fact that those lines of the program aren't going to be there later. They are: a) no function statements may be defined there, no ONERR or ONEOF statements may be defined or have their destination there,

d) no common or module code may refer to any line number here after the first BLOAD, and c) any string assignments must force the string contents out of the program and into normal string storage. The approach below will do the latter:

```
READ A$: TITLE$= A$ + ""
B$= "This is a string" + ""
```

The string concatenation forces BB to move the string out of the program area. Any other string function, such as MID$, could also be used.

In our program, the common code runs through line number 997. Line 998 is the transition subroutine, and code just above that point does the actual overlay. Each proto-module is created as a BB program. You may choose to write it following the common code to permit testing.

When ready to save the module as a binary module, SAVE the program first, then EXEC the following text file:

```
DEL 1, 998: REM Remember, modules
can't have these low line numbers
800 INPUT "MODULE NAME TO SAVE?
";A$: FILE$= "MODULE."+A$
810 GOSUB 998:
last%=exfn%.peek(%61,%0,%1): len%=
last% - addr%
820 PERFORM
bsave(@FILE$,%addr%,%xtend%,@len%)
830 PRINT FILE$;" contains
";len%;" bytes.": END
998 addr%=exfn%.peek(%79,%0,%1):
xtend%=exfn%.peek(%5712,%0,%0):
  addr%=addr%+exfn%.peek(%addr+1,%xtend,0):
RETURN
RUN
```

Line 998 calculates the address of the line following it. The peek in line 810 reads the BB end of program pointer. Thus, each module will begin at the line following 998, and the length written is (as calculated in variable "len%") the number of bytes between this point and the end of the program.

In BB it does not matter (as it emphatically does in Applesoft) that the common code be in memory, and always of the same length, when the binary files are written or read. This is because BB uses relative next-line pointers. That is, each line of BB code begins with a two-byte pointer containing the number of bytes to add to the line pointer in order to find the next line. In Applesoft this pointer contains the absolute address of the next line. The happy result of this design improvement in BB is that you may modify the common code to your heart's content after the modules have been saved.

It is a help if the actual execution code in each module begins with the same line number. The common code can then execute that module by a simple GOSUB 1000, or whatever. If more than one command is contained in a module, then you'll have to keep track of the beginning line numbers of each command, and use an ON k GOSUB ln1,ln2...,ln3 statement. Of course, you also have to know the name of the module in which each command was BSAVEd, and when a command is given by the user, BLOAD that module if it is not now present.

The module code must follow the rules given for the initialization code, above: no embedded strings that are used outside of the module, and no

lines or functions called by other modules.

The code to load in a module is similar to that above:

```
800 PRINT"LOADING MODULE ";A$:
FILE$= "MODULE."+A$
810 GOSUB 998
820 PERFORM
bload(@FILE$,%addr,%xtend,@len%)
830 RETURN
998 addr%=exfn%.peek(%79,%0,%1):
xtend%=exfn%.peek(%5712,%0,%0):
  addr%=addr%+exfn%.peek(%addr+1,%xtend,0):
RETURN
```

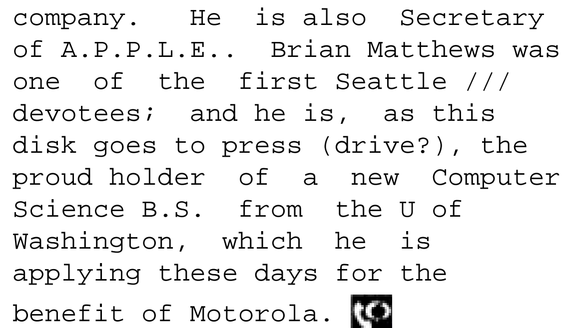
When ready to load a module, GOSUB 800 with A$ equal to the last portion of the module file's name. Then GOTO or GOSUB to the code in that module. In use, the modules load very quickly, and of course, there is much less space taken up by the program code. Try it.

THE AIDA.TOOLS INVOKABLE

The assembler source code for AIDA.TOOLS is found in another file on this disk. The ready-to-use invokable is in its own file elsewhere on this disk.

Note that the authors have provided AIDA.TOOLS for the individual use of our readers. Commercial rights are, however, reserved.

///   ///   ///

Author Bibliograhies: Dave Lingwood sports the checkered career typical of many in microcomputing: a background in English and Journalism, PhD in Communication Research, and R & D work in technology transfer and marketing, before forming his own software/information service company. He is also Secretary of A.P.P.L.E.. Brian Matthews was one of the first Seattle /// devotees; and he is, as this disk goes to press (drive?), the proud holder of a new Computer Science B.S. from the U of Washington, which he is applying these days for the benefit of Motorola. 

SETDEBUG

Brian Matthews

MONITORING THE MONITOR

One of the greatest advantages of the Apple][was the "openness" of the machine, and one of the key factors in this was the existence of the Monitor, that allowed someone to dig around directly in memory. By simply pressing the reset key, you could look at your machine language programs, or the disk operating system, or even the Monitor itself. Of course this wasn't the perfect way to do things, you couldn't hit reset with a program running, examine or change things in the Monitor, and then reenter Applesoft and have the program continue running where it left off. You had to restart the program from the beginning.

Then along came the Apple ///. Sure enough, the /// had a Monitor just like its older brother the Apple][, and by holding down the Control and Open Apple keys while pressing reset, there you were, in the Monitor. Again you could examine memory directly, and change things, but wait, how do I get back to what I was doing?? Well, according to Apple, and most everyone else, it couldn't

be done. Once you performed a reset and were in the Monitor, the only way out was to reboot a disk, and technically, what everyone said was correct. To see why, we have to look at what a reset really does.

When you hold down the Control key, and press reset, this sends what is known as a reset signal to the 6502 microprocessor. It tells the 6502 to stop what it's doing, clean up all of its internals, and start executing the machine language program at a certain prespecified place in memory. In the Apple /// this is the diagnostic portion of the Monitor. When Control and reset are pressed, the Monitor sets up some locations in memory for itself, sets the machine state, and performs some diagnostics to make sure the machine is working properly. Then it checks to see if the Open Apple key is pressed. If not, the Monitor attempts to boot the disk in the built-in drive. However, if the Open Apple key is pressed, the Monitor starts accepting input from the keyboard, and executing commands. This is what is generally known as the "Monitor", though everything is actually part of it.

The problem comes though, because by the time you can actually type commands to the Monitor, things are so messed up by the Monitor and its diagnostics that there is really nothing to return to, so you are stuck in the Monitor until you reboot.

At this point, things look pretty bleak. By the time you actually get into the Monitor, SOS and the program you were running are pretty messed up, so there is no way to do a reset, get into the Monitor, and then return. It seems that when everyone said it couldn't be done, they were right.

So now that we have decided it couldn't be done using reset, let's look somewhere else. So far, we've been using Control and reset together. What happens if you just press reset? In this case, a reset signal isn't generated, instead an NMI signal is generated. An NMI is a Non-Maskable Interrupt, or just a fancy way of saying an offer that the 6502 can't ignore. When an NMI occurs, the 6502 saves everything, so later it can return to what is was doing before the NMI occurred. After everything is saved, the 6502 starts executing at another prespecified location. This time though, the machine language that gets executed isn't in the Monitor, but in SOS itself. However, the routine consists of one instruction, and RTS, which tells the 6502 to go back to what it was doing earlier. In other words, an NMI doesn't do anything. (This isn't entirely true. Some routines in SOS get executed first, but the net result is as if just an RTS was executed.) To prove this to yourself, boot Business Basic or Pascal, and press just the reset key. It appears that nothing happens, but you now know that the 6502 receives an NMI signal, saves everything, executes and RTS, and returns to Basic or Pascal. (This isn't entirely true either. Business Basic version 1.1 and earlier replaced some addresses in SOS with some of its own, so when you pressed reset, the routine that was executed was in Basic, so pressing reset was effectively the same as pressing Control-C. In Business Basic versions 1.2 and later, this isn't done, but it is

an interesting concept that we'll come back to shortly.)

So now that we know the difference between a reset and an NMI, it seems that an NMI is what we want to use to get into the Monitor and back. The problem though, is that the address in SOS that points to the routine to be executed points to an RTS. The solution? Why, modify SOS of course. With a little digging around, it's possible to find where the address of the NMI routine is. It is part of a JMP instruction at $1910, so the address sits at $1911 and $1912, with the low part of the address in the first location, as is standard with the 6502. We can have an NMI execute anywhere by simply storing the address of our routine at $1911 and $1912.

The problem now becomes, what do we want to point the NMI at?? The first thought may be somewhere in the Monitor. Unfortunately, the Monitor requires the machine to be set up in a certain way, and if we enter at the wrong place, the machine won't be set up, and the Apple will die a miserable death. (Of course the machine itself isn't damaged, but you would have to reboot.) If we enter early enough to set things up for the Monitor though, all the diagnostics end up being executed, and we're no better off than if we had done a reset.

Again things start to look bleak. We need a short little bit of machine language that sets things up for the Monitor, enters the Monitor, and then restores stuff and returns to executing the program that was running before the NMI. In this case, we're lucky because it seems that the good folks at Apple wanted to just the same thing we're trying to do, so they placed a routine in SOS to do just what we want. The entry point to the routine is located seven bytes behind the RTS that is normally executed, so we just have to subtract seven from the address normally at $1911 and $1912, and store the new address back in $1911 and $1912

And that is just what SETDEBUG does. It actually gets the address of the NMI RTS from $1904 and $1905, so if SETDEBUG is performed twice, the second time won't reduce the address by another seven bytes. SETDEBUG simply subtracts seven bytes from this address, and stores the new address in the JMP instruction that jumps to the NMI routine, so you can now enter the Monitor and get back out. To enter it, just press reset by itself. To return, type 198CG and press return. This starts the second half of the routine executing, that restores things and returns from the NMI.

THE PROGRAM

To actually perform SETDEBUG, from Basic type INVOKE .D1/SETDEBUG.INV, or whatever you've called it. Once it has been invoked, type PERFORM SETDEBUG and press return. That's all there is to it. From Pascal type X to execute a program, and tell Pascal to execute .D1/SETDEBUG (or again whatever you've called it). The program will be executed, and you will be returned to the Pascal command line. In either case, you can now press reset (remember, just reset and not Control reset) and be in the Monitor.

Now that we can get into the Monitor and back, what can we do? The first thing you may want to

do is to press Escape, and then 8, to clear the screen and set 80 column mode. If you don't, the Monitor won't handle the screen correctly, being it thinks you're in 40 column mode, but the hardware is displaying 80 columns. Everything will work the way it should, it will just be a little difficult to see.

Below is a list of all of the Monitor commands. Perhaps two of the most interesting are the R and W commands. You can actually read in blocks directly off of the disk, something the Apple][Monitor can't do.

Also, you can now interrupt a running program by entering the Monitor, and then when you return, the program will continue running! This is because the NMI is treated like a subroutine, and the program that was interrupted doesn't even know it was interrupted. To try it out, enter this Basicprogram:

```
     10   FOR x=1 TO 10000
     20     PRINT x
     30     NEXT
     40   END
```

After you've entered it, run it, and after it has printed a few screens full of numbers, press reset. (Of course you must have invoked and performed setdebug earlier.) You will then be in the Monitor. Press Escape, 8 and return to clear the screen. Then type 1990.19CF and hit return. This prints the contents of memory locations 1990 to 19CF to the screen, both as hexadecimal digits, and as text. You should see that these locations hold a copyright notice for SOS. Now type 198CG and press return. This returns you to the program you were running, and it should continue printing numbers to the screen, right where it left off!

You can do the same thing from Pascal. Enter and compile this program:

```
    program test (input, output);
    var
      count : integer;
    begin
      for count := 1 to 10000 do
writeln (count);
    end.
```

After it has been compiled, and you have executed SETDEBUG, execute the program, and hit reset after it has printed a couple pages of numbers. Do an Escape 8 Return, and then type 1990.19CF. Then type 198CG and the program should continue printing numbers where it left off!

One word of warning. Because you can modify almost any byte in memory, and SOS, Basic and Pascal are all stored in memory, you can easily stomp on something you shouldn't. Also, you can write to any block of a disk, easily overwriting something important. The moral is, BE CAREFUL. Don't have anything in memory or on the disk in the built in drive that you can't afford to lose. Other than these warnings, go to it.

MONITOR COMMANDS

Here, ADDR1, ADDR2, and ADDR3 are 4 digit hexadecimal addresses. BYTE1 and BYTE2 are 2 digit hexadecimal values. BLOCK is a physical block on the disk, in hexadecimal. Return must be typed after each command. Commands that don't modify memory can be separated by spaces, so typing 198C FFEF 1901 will print the contents of location $198C, $FFEF, and $1901, respectively. Commands that store

values in memory must be separated by a /, so typing

198C:4C/FFEF:00/1901:06

will store $4C in location $198C, zero in location $FFEF, and 6 in location $1901, respectively. When an address or range of addresses is dumped, they are printed both as hexadecimal values and ASCII text. If the present mode is 40 columns, 8 values per line are printed, otherwise 16 are printed if in 80 column mode. The ESCAPE commands consist of hitting the Escape key, and then the specified key, so to clear the screen, you would hit the Escape key, and then the S key. You should then hit Space to get out of Escape mode.

Command:	Meaning:
ADDR1	Dump the byte at ADDR1.
ADDR1.ADDR2	Dump the bytes between ADDR1 and ADDR2.
Return	Dump the next line of bytes.
ADDR1<ADDR2.ADDR3M	Move the bytes at ADDR2 through ADDR3 into ADDR1
ADDR1<ADDR2.ADDR3V	Verify that the bytes between ADDR2 and ADDR3 are the same as those at ADDR1.
BYTE1<ADDR2.ADDR3S	Search for BYTE1 between ADDR2 and ADDR3.
BLOCK<ADDR2.ADDR3R	Read the blocks starting at BLOCK into ADDR2 through ADDR3.
BLOCK<ADDR2.ADDR3W	Write to the blocks starting at BLOCK from ADDR2 through ADDR3.
ADDR1:BYTE1 BYTE2...	Store BYTE1 at location ADDR1, BYTE2 at location ADDR1 + 1, etc.
ADDR1:'Text'	Store the Text into memory starting at ADDR1 with the high bit of each byte clear.
ADDR1:"Text"	Store the Text into memory starting at ADDR1 with the high bit if each byte set.
U	JSR to location $3F8. (Locations $3F8 through $3FA should have a JMP instruction to a machine language routine stored in them.)
ADDR1G	JSR to ADDR1 returning to the Monitor when done. (ADDR1 should be the address of a machine language routine.)
ADDR1J	JMP to ADDR1. This never returns. (ADDR1 should be the address of a machine language routine.)
<space>X<space>	Repeat the previous command on this line until space is pressed. The X must have a space before and after it.
ESCAPE 4	Set 40 column mode and clear the screen.
ESCAPE 8	Set 80 column mode and clear the screen.
ESCAPE L	Clear to the end of the current line.
ESCAPE P	Clear to the end of the current page.
ESCAPE S	Clear the screen.
ESCAPE ARROW	Move the cursor in the direction of the arrow.
SPACE	Pause a memory dump, then print a single line at a time. Any other key except TAB continues at normal speed.
TAB	Stop the current command being executed, return-

```
.title          "SETDEBUG, point NMI's at SOS' debug routine"

.list
.nopatchlist
.nomacrolist

.proc           setdebug

;Setdebug points SOS' NMI vector at the debug routine in SOS.  It normally
;points at an RTS so that hitting RESET doesn't do anything.  Setdebug
;changes it so when you hit RESET, SOS enters a routine that saves all the
;important stuff, and jumps into the built in monitor.  To reenter SOS, do
;a 198CG from the monitor.  Known to work through SOS 1.3, and may work
;in higher versions.

lda             1904            ;Grab low byte of NMI vector
sec                             ;Make sure that carry's set.
sbc             #7              ;Fall back 7 bytes from the
sta             1911            ;byte currently pointed to
lda             1905            ;(an RTS), and store this in
sbc             #0              ;the NMI JMP instruction.
sta             1912            ;Unwrap that darn high byte.
rts                             ;That's it!!!
```

A SIMPLE LIST MANAGER IN PASCAL

Mike Christensen

This program provides a way to create, examine, edit, store and retrieve a simple list. Lists can grow to be as large as available memory will permit, but the List Manager program is only efficient for shorter lists. Too trivial to be a serious application tool, List Manager was meant to be a foundation for programmers who want to use linked lists in their applications.

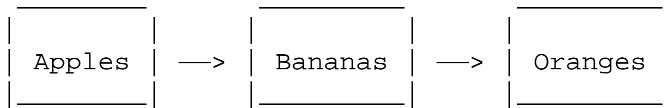
Linked Lists In Pascal

Many application programs need lists where the number of items in the list is not known. The programmer could use an array, but that would mean guessing at the largest number of items that would ever appear in the list. When combined with the dynamic variables supported by Pascal, the simple concept of a linked list provides a useful solution.

There are a number of ways to arrange an ordered list in memory. For large lists (perhaps a hundred or more items) where being able to quickly locate certain information is important, separate indexing schemes are often appropriate (B-Tree, binary tree, et.al.). But for smaller lists, simply linking each item to the next is adequate. Each item can have an embedded pointer to the next item. This is called a 'simple linked list'.

(see Table 1)

Table 1



To see any single item in a linked list, the list must be 'traversed', starting at one end, and moving to the other until the item is encountered. It can only be traversed in one direction. Operations like inserting and deleting items in the middle of the list are relatively simple to perform.

By adding a 'back pointer' to each element, the list can be traversed both forward and back. This is often quite useful, giving the ability to move back a single element without traversing the entire list. This kind of list may be called a 'double-linked list'.

(See Table 2)

Table 2

Another useful enhancement to a linked list is to let the last item point forward to the first item. This way, when the end of the list is reached while traversing, the next record is the first. This allows the traversal to begin in the middle of the list and still be able to traverse all records in the list. This closed list is said to be "ring-structured".

(See Table 3)

Table 3

If you use linked lists in your programs, it is important to weigh their strengths against their limitations. Linked lists are efficient for inserting and deleting when the list is small. If a linked list is dynamic, then it will use only as much memory as is required, unlike a static array. Linked lists are very slow for sorting, and very inefficient for searching when the list grows large. Accessing an element at the end of the list may require traversing past all the elements before it.

The List Manager Program

The List Manager program that accompanies this article is menu driven. Just type the first letter of the option you want; RETURN, SPACE, and ESCAPE are used in much the same manner as the p-system. The following functions are supported:

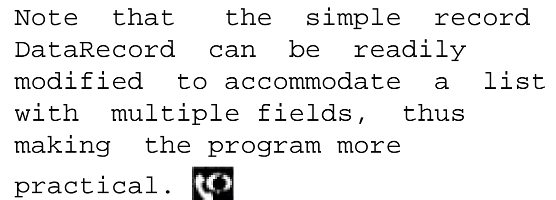
- Add an entry to the list
- Change
  - Insert before an entry
  - Insert after an entry
  - Insert in ascending order
  - Insert in descending order
  - Exchange an old entry for a new entry
  - Delete an entry
- Sort
  - Ascending (A to Z)
  - Descending (Z to A)
- Read a list from disk
- Save a list on disk
- View
  - Forward in arrival sequence (first to last)
  - Backward (last to first)
  - Ascending (A to Z)
  - Descending (Z to A)
  - Count of entries within the list
  - Length of longest entry within the list
- Quit leaves that program

Most applications won't need all the functions. In the source, most functions are isolated so you can remove those you don't use.

When sorting and viewing, it is important to realize that keys are handled in a case sensitive manner; thus, uppercase keys will appear ahead of lowercase:

Able
Baker
able
baker

Note that the simple record DataRecord can be readily modified to accommodate a list with multiple fields, thus making the program more practical.



Notes:

BUSINESS BASIC:
A UTILITY FROM THE LAZY H

Bob Huelsdonk

I'm lazy. Not ordinary downright "lazy" lazy, mind you, but just lazy enough to not want to clutter up my mind with a lot of routine words that I must remember in order to get certain things accomplished on my Apple ///, and just lazy enough to not want to type ten or fifteen characters when one should do. I will defend my laziness with proper zeal by proposing that it leaves my mind free for all sorts of much more valuable stuff such as the latest data on llama importation into the U.S., but I stray from the subject at hand.

I wrote a short program which helps do a number of the routine chores in program development. I wrote it so that it could be easily added to an existing BASIC program and easily deleted when you are through with it.

The program accompanies this article on this, the first edition of the /// CHEERS disk "magazine." It is about 6k bytes long and is provided in its BASIC form. That is, a normal BASIC program as opposed to the alternative of saving it as an "EXEC" file. It is handy to have it in EXEC form, though. So, I'll show you how to use the utility program to make an EXEC of itself.

First, some introduction: For the uninitiated, an "EXEC" file in this case is a BASIC program listing saved as a TEXT file (in this form it uses about 8k bytes on the disk). When this is done, the program segment (just a piece of a total program) can be added to an existing BASIC program by the command:

```
)exec /basic/util.ex
```

This command would cause SOS to look for a diskette named BASIC in one of the configured disk drives and would then look for a TEXT file on that diskette named UTIL.EX (this is the name that I use for this utility program: UTIL because it is my utility program, "." for a separator, and EX to denote an EXEC file) and would then write the text file to the console. The file does not display the lines to the screen but does show a ")" prompt for each line read. In doing so, the BASIC interpreter does not know that you have not suddenly become an extremely fast typist and thinks that you are entering the lines from the keyboard. Voila! You have added the program segment to your existing program in memory.

Note that I purposely show the command line in lower case. The reason for this is that I do all BASIC programming work in lower case. The system does not care and in the case of program lines, all valid commands are converted to upper case when the program is listed and so you have a convenient check by scanning for lower case variables and upper case commands.

I mentioned that this disk has a copy of the utility as a BASIC program. That is because the utility has within it a routine to make itself into a TEXT or EXEC file. We shall discuss this a little later. The copy on the disk also has six lines which are added at the beginning of your existing program. Four are handy and two are necessary to it's proper operation unless you choose to change this feature. These lines are:

```
1  GOTO 10:REM Pointer to user program beginning.
2  :
3  pgm.name$="UTIL":RETURN
4  :
5  GOTO 60000
6  GOTO 60008
```

Lines two and four are not necessary and are just for show to set off the program name. Line 3 is to identify the name of the program. You would substitute the actual name of the program you are working on instead of "UTIL." I chose line numbers less than 10 since program listings which might preexist and to which you might want to add this utility usually start with 10. The bulk of the program uses lines above 60000; again to avoid usually used line number ranges so when you EXEC the segment on a program you will not be likely to get duplicate line numbers. If you did, the new lines would replace the preexisting lines and part of your previous program would be overwritten. Of course you had a back-up, right? Line 5 is my laziness showing again. I am too lazy to type "goto 60000" and so I only need to type "goto 5." Line 6 is another quick feature with a goal of using less total disk space.

When I first developed this utility, I added it to each program I wrote because I really missed it if it weren't there. After awhile I must have had it stored at least a hundred times on various diskettes and I am too Scotch to use that much space redundantly. Also I wrote very tight and uncommented code to save space. I recently modified the utility so that I can easily delete most of the code and still have a program "save" routine available. Then when I want the full utility available, I type "goto 6" and the utility is automatically restored to the end of the program if my /BASIC/ diskette is in one of the drives. I use three drives, a luxury we don't all have but it can be done nicely with two. However you must change diskettes to make duplicate back-ups, a habit I would encourage.

The line number one shown is to allow normal operation of your program with a "run" command. It would "GOTO" whatever line is the normal first line of your program. There is one more change to the program you should make to allow it to work properly and that is to identify the number of disk drives on you system. Line 60004 has a variable num.drives=3" in it. If you only have two drives on your system, edit this line by the usual cursor trace over method to change the number to two.

An Overview of Utility Operation

After you have added the utility to your program (or started out with it alone, including line 3), you use it by typing "goto 5" and you will get the following prompt:

```
     (p)rint     (d)isplay
(f)ile:
```

It does not matter if you had moved the cursor up to the middle of the screen and had program text all around you because the PRINT CHR$(29) in line 60004 clears the screen from the command line down. The menus are hierarchical. This is a highfalutin word that means one menu leads to others. There are just three layers in this utility structure. This first command line points to three different kinds of activity. Printing information to the printer, displaying information to the screen, or filing information to diskettes. The second layer gives choice menus for these three activities, and the third layer prompts for information as needed. Some special cases of these will be discussed.

The Print Functions

In order to follow the rest of this article, it would be handy if you had a printed listing of the utility. And you may just as well get that listing by using the utility itself. You do this by the following commands:

```
     )load /cheers.1.1/util
     )goto 5
     (p)rint     (d)isplay
(f)ile:p
     (l)ist      (c)at
(r)emarks   (g)o refs   (t)ext
file:l
      OUTPUT#9:LIST
:?#9;PR$:GOTO 60450
```

You will have noticed that just before the last command line is displayed, something was printed to your printer. (The program does assume the printer is on and paper loaded.) The line printed was the program name, the date, and the time of day. If you haven't added a real time clock yet, shame on you. Now you see why the program name is required and the date and time stamps do wonders for tracing through program development. When you reach this point, the cursor will be positioned over the "O" in OUTPUT at the beginning of the last command line. This lets you to input data into the command. Namely the line range desired. Otherwise you would always print out the entire listing.

You perform this entry by using the right arrow (without pressing "ESCAPE") to trace to one space past the "T" in LIST. Then line number ranges can be entered in the form "LIST xxxxx-yyyyy." In this case, you want the whole listing, so you can ignore inputting the line numbers and just trace over the whole thing. The remainder of the command is than traced over and at the end (just past the 0 in 60450) press carriage return.

Congratulations!! You have just used UTIL for the first time. Or did something go wrong. The program does assume a printer driver named ".PRINTER" and sends control characters for an EPSON in conjunction with a PKASO card to give listings with perforation skip over. This is done with the pr$=CHR$(2)+CHR$(27)+CHR$(79) in line 60200. If this gives you trouble, you can change the line for your printer or set pr$="" in line 60200 for now to get you going. If all went well, you now have a printout to follow.

Since we have started by using the print commands, lets look at the rest of them. You may have

noticed that commands require only one character and no carriage return is required. If you enter a menu by mistake or accident you only have to input a carriage return to return to the main menu. If you are at the main menu, a carriage return will drop you out of the utility and into the BASIC operating system. The second printing command "c" will print the catalog listing from the diskette with the presently selected prefix to your printer. The third command "r" will result in a prompt:

TEXT:

This will allow you to print remarks to the printer if you wish to add comments to a listing, to a catalog listing, or just to make a few notes to remember. This is used by typing a message line just after the TEXT: prompt. The message will show on the screen and can be edited by using the cursor. When you press the carriage return, the message line will be printed to the printer. One thing to watch out for; word wrap-around is not automatic except as your printer will control it so make your lines 80 characters or less. You can keep entering as many lines as you wish with a carriage return at the end of each. When you are through, a single carriage return at the beginning of the input request will bring you back to the main menu.

The next print function is the "(g)o-refs" which simplifies the use of the renumber program furnished by Apple to print a list of line references for your program. This is very useful when you are developing a program or to analyze the operation of an existing program. This function assumes that you have at least two floppy drives on your system and that the /BASIC/ diskette with the RENUMBER.INV invokable module is in drive one.

This function warns the user that it will read the latest version of the program from drive two and will list the line references to the printer. You are prompted to proceed by pressing "y." Any other input will abort the function. This function requires that the "RENUMBER.INV" invokable module be invoked. The utility program will automatically invoke this module if it is not already. If you wish to configure this function differently, then you can change the diskette name in line number 60380.

The last print function is (t)ext file and allows you to print any text file that is in one of your drives to the printer. If the file is on a diskette that does not have the selected prefix, then you either need to change the prefix selection (see how later) or enter the full name of the catalog and file name such as:

File name? /BASIC/NOTES

The file will then be printed to your printer.

The Display Function

A "d" entered from the main menu will show the display menu:

(t)ext file (l)abel lines (h)elp file:

The first function "t" will prompt for the name of a text file. The same rules apply as in the above paragraph referring to the printing of text files. In

this case the file will be displayed to the screen. If the file is longer than twenty lines, you may use CONTROL number-pad 7 to halt the file scrolling at any time to read sections of it.

The next function "l" is to help be a little neater in labeling sections of the program. I like to have good looking listings with easily discerned title blocks for both main routines and sub-routines. You can see the results of this labeling by looking at the listing of this utility program. Lines 60187 to 60189 illustrate a main heading and lines 60197 to 60199 illustrate a sub-routine label. When you use this function from the utility program you will get a display of partial line numbers and untitled label lines. You fill in the uncompleted information by tracing over symbols you want and filling in what is needed. If you are doing a main routine, type a hyphen over the vertical bar in the middle of the long lines. If you are doing a sub-routine label, press carriage return at the vertical bar.

The final function on the display menu is (h) to display to the CRT, a help file of information as a handy programming reference. The help file is just a TEXT file with the desired information in it. It can be created using any word processor or editor which will produce an ASCII TEXT file. If there is enough interest, I will submit a help file sample for a future issue. The program assumes that the help file is on the /BASIC/ diskette in one of the drives and that the file is called "HELP."

The Filer Functions

There are six functions accessible from the filer menu:

```
     (d)rive      (e)xec
(s)ave      (-)save   w/o   util
(r)enumber
(i)nvoke
```

The first command "d", results in the following prompt:

```
     Init drive #:
```

The result of inputting a number is to select the prefix of the diskette in that drive. This is handy if you are going to be working with one drive primarily for awhile or to eliminate having to type in the prefix each time you want to change it.

The second command "e" allows you to easily make an EXEC file from any part of your program. You will first see the prompt:

```
     File name?
```

You must type in the file name you wish to use for the EXEC file. You must include the pathname of the diskette if you wish the file to go to other than the diskette with the presently selected prefix. You will then see the following command line:

```
     OUTPUT#8:LIST
:CLOSE#8:GOTO 60000
```

Again the cursor will appear over the "O" in OUTPUT. You trace over the command line just as you did for program listings, adding the line number range you wish included in your EXEC file as LIST xxxxx-yyyyy. Then finish tracing over the line and your EXEC file will be saved to the diskette.

This is the function which allows you to save this utility as an EXEC file. You would enter the name "/basic/util.ex" to the file name prompt and would enter "LIST 60000-60999" in the command line. Note that you do not save the lines from 1 to 6 as this would write these erroneously to another program that you added this utility to.

The third function is selected by "s" for saving your program to disk. After you press "s" you will see the following prompt:

Save UTIL to drive #:

The program name "UTIL" will be replaced with whatever you have named your program in line three. You then input the number of the drive to which you want the program saved. The program will first delete any copy of the program which is presently on the diskette to reduce fragmentation and to eliminate any possible errors from having reduced the program size. If the program did not previously exist on that diskette, then the bell will ring and the following message will appear in inverse:

NEW to this DISK

After the program has been saved to the disk, the directory is given a rudimentary check. I added this because earlier versions of SOS had bugs which allowed occasional overwriting of files on other files. I believe there are still occasions where this will happen. If you feel confident, you can eliminate this step but it does not take long and I have had it save a lost disk because if I get a "BAD CAT" I do not save to the second diskette until I have solved the problem. The check is done whenever you select a different drive or when you save a program. It works on all floppy diskettes except those with sub-directories which are labeled "CAT" on the catalog listing.

If the directory is correct, you will see "CAT OK."; if there has been directory damage as determined by the blocks not tallying, then you will hear the bell and the message "CAT BAD!" will be displayed. After the program has been saved and the directory checked, you will be prompted to save it again (to another diskette). If you chose not to, or if this is your second save anyway, a carriage return will bring you back to the main menu.

The fourth command "-" will save just as the previous one did, however before it saves the program it will delete the utility program except for lines 60000-60008. This is to save diskette space as mentioned before. Remember that you can re-add the full utility program whenever you wish with the "goto 6" command and you can save the program with the remaining short portion.

The fifth command "r" for renumber is one of the more complex commands. The renumber program supplied from Apple Inc. operates by reading a disk file, performing the renumbering and then saving back to another disk file. This can be done by using two different file names on the same diskette but I chose to use the same file name on two different diskettes. If you do not like this approach an examination of lines 60800 to 60840 should help you to rewrite the way you choose. The renumber program is an invokable module and must either have been invoked or must be on a disk named "/BASIC/"

in one of the drives. If you have only two drives, you can use a disk named "/BASIC/" that has the "RENUMBER.INV" module on it as one of your development diskettes or you can invoke the module from your basic diskette before you enter the renumber function (you can use the next described function to assist in the invoking). I will show how the prompts would look if you use three drives:

/BASIC/ must be in a drive or RENUMBER.INV must be invoked. Read from drive .d2 & renumber to .d3, then load from .d3 (y)es?

```
Oldstart Oldend Newstart
Newincrement
      PERFORM resequence(@i$,o$,%0
,%59999,%10   ,%10   ):GOTO 60840
```

As before we use our old trick of tracing over a command with the cursor to adapt it to our needs. If you trace it over just as it stands, you will renumber all lines from 0 to 59999 by increments of 10 and the new first line will be 10. Usually, however, you will not want to renumber the entire program. Thus you can enter and control just how the renumbering will operate. You do this by entering per the prompts printed just above the variables in the command line. Oldstart is the first line in the existing program that you want to renumber. Oldend is the last line of the existing program that you wish to have renumbered.

Newstart is the beginning line number that you wish your first new renumbered line to have. Newincrement is the increment which will be used for the renumbered lines. If you have a collision of new line numbers with existing line numbers the renumbering will stop and an error message will be given.

After the lines have been renumbered, you have a copy of the old numbering on the diskette in drive two and a copy of the newly numbered program in drive three. The program then loads the newly numbered program to the console so that you can inspect it. If some problem has occurred, you can start again from the original in drive two. If all is O.K. and you wish to retain the new copy, then you can save the console copy back to drive 2 and you will have two fresh copies to work from.

The last function in the filer section is the "i" for invoke command. This is not greatly needed if you keep the /BASIC/ diskette in one drive and use ON ERR to automatically invoke the modules but if you have fewer drives, you may find it convenient to pre-invoke the modules that you might need. After pressing "i" you will see the following prompt:

```
     (r)eadcrt   (n)renumber
(g)raphics  (c)omb r&n (b)oth r&g:
```

Inputting the proper letter will invoke the modules as shown presuming that they are on a diskette called "/BASIC/" in a drive. You may change the modules or combinations of them by editing lines 60910 to 60950 and then changing the prompt and command letters "rngcb" in line 60900. If you are not familiar with this technique, it simply uses the command letters in an "ON INSTR"ing function to GOTO the selected line numbers.

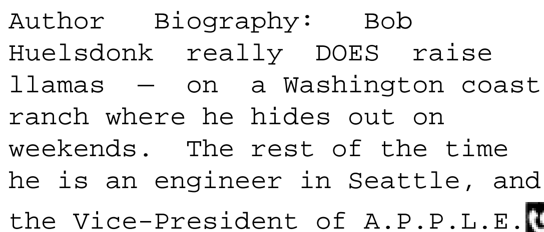
Program Summary

BASIC line numbers used:
1,3,60000-69999
File numbers used:
8,9
Memory required:
6k bytes
Hardware:
Clock, two disk drives, printer
Printer driver name:
.PRINTER

That’s it! I hope the program can be of some real use to you. If you find additional functions or clever changes, you might submit them to our editorial department for inclusion in a future edition of “/// CHEERS.”

/// /// ///

Author Biography: Bob Huelsdonk really DOES raise llamas — on a Washington coast ranch where he hides out on weekends. The rest of the time he is an engineer in Seattle, and the Vice-President of A.P.P.L.E.



Notes:

VERSAFORM: A Closer Look

Terri Freeman

We Apple /// owners have waited patiently for the fulfillment of Apple’s promise of software for the Apple ///. We always have heard that the Apple /// is a powerful, wonderful machine whose potential had barely been tapped; but a user practically had to be a programmer to realize this wonderful potential. Now, with the production of VersaForm, by Applied Software Technology, we have a program that does a superior job of such applications as invoicing, client billing, and inventory control. Its performance is more than worthy of the Apple ///’s capabilities.

VersaForm is advertised as the database that fits itself to the forms you already use — the invoices, purchase orders, personnel or client records — any form you use now can be reproduced into the VersaForm format. You fill in the form on the screen, and VersaForm will print out directly onto the same forms you use.

A CUSTOM FORMS DATABASE

You set up VersaForm to suit your own forms; you customize a screen layout for entering data in the way you want, and you customize the output or form printing in the way you want. You no longer have to buy invoices or records that are printed to fit your software — you can use the invoices you have right now, or design your own!

The first thing you do is design a form, or set up the screen layout. But VersaForm provides a step-by-step tutorial to teach the new user how to use VersaForm, so you don't have to actually set up a new form just to learn how to use the program. Sample files that have been set up help the user become familiar with VersaForm's functions, without having enter their own data into a completely new program. This is the best way to learn what a program can do when you start using your own data.

THE VERSAFORM MENU

These are the five functions of the VersaForm program.

1. Design a Form (screen layout)
2. Filing (enter data)
3. Report (analyze data)
4. Design a print format (for printing individual forms)
5. Copy or print forms

ERROR CHECKING ROUTINES FOR CORRECT DATA ENTRY

VersaForm is designed to emulate your present forms, and it goes one step further: it has a set of checking routines to look for errors in data entry. So when the form is filled in, the checking routines make sure that most of the data is correct. These routines include left- and right-justification, whether the data must be numeric, whether the entry must be one of a set of acceptable answers, what a minimum or maximum allowable entry is. Like VisiCalc, an entry may be calculated from one or more other entries. If the entry is usually some specific constant, that can be entered automatically. In each of the automatically entered items, the entry can be overridden by a manual entry, which is important to allow data entry flexibility. Suppose the price of an item on your form is almost always $3.00. This week you have a sale on that item, and it's $2.29. You don't have to change the error checking built into your form, just enter the price manually.

Probably the most unique and attractive error checking routine is the lookup tables. You enter a two part list, consisting of the entry to look for in one data item, and the corresponding entry to insert in another data item. We use the lookup tables to fetch the price of an item, the description of an item, and the name and address of a customer. You are allowed up to 99 entries for each data item. Although this may seem a strict limitation, in many applications the lookup function is invaluable, and the 99 item restriction is quite ample. Suppose you have 500 or 600 different customers — you can enter the 99 names you sell to most frequently. That means that for those 99 customers, VersaForm can automatically fill in such data items as Account Number, Address, Type of Transaction (Wholesale or Retail), Discount %, and the like. Furthermore, for those customers not included on the lookup table, you could enter the Type of Transaction from the keyboard. VersaForm will automatically fill in the Discount % and calculate prices, for instance, if you have set it up to look up values based on the Type of Transaction. This makes VersaForm different from almost any other database available.

DESIGNING A FORM

First you design the screen layout of a form, then for each item you enter the error checking routines you want, using the “Design” function. Those routines can be changed at any time, but the screen layout can’t be changed after you have entered records. There is a special copy program which comes with VersaForm, however, that allows you to copy records from one form into a new form, matching the names of the data items. First you have to design the new form, then copy the records into it.

Mailing List

Old File	New File
Name	Company
Address	Name
City/State/Zip	Address
Phone	City/State/Zip
	Phone

In the above example, records from the old file can be copied into the new file (to which the data item “Company” has been added) by using the “Copy by Name” program which comes with VersaForm. That means you can add or delete data items at any time. However, you cannot easily change the length of a data item — the length of a data item must be the same in the old file and in the new file.

To change the error checking routines you must go back into the design function of the program. One caution: it is a little awkward to go in and change the error checking routines on a file. You have to go through each data item sequentially until you reach the item or items you wish to change. This can be minimized by careful planning when you first design the form.

ENTERING DATA

Once a form is designed, you use the “Filing” function to enter records into the file. When the records are saved, they are indexed according to the entry in their “Key” data item — for example, in a invoice it might be the invoice number. You can access a form to change it only by calling it up by its key entry. The key may be made up of one data item, or two data items combined. This is a very limited method of access, but you can organize your records so they are easy to access using this key. When you print a report to analyze your data you can select records according to any of your data items, not just the key item. You can also print records like an invoice based on any of the data items, using the “Copy/Print” function.

PRINTING ROUTINES: REPORTS AND PRINT FORMATS

There are three ways the VersaForm program can print records: with a report, using a print format, and without using a print format (a copy of the screen).

The report prints out data from the records you select, sorting, totalling numeric data and other features. It will even include a second file (of the same form) in

the same report. The report can be printed to the printer, to the screen, or to a Pascal text file which the user can edit separately. The report rules the user chooses are always saved to be used again later. If you want the program to ask you for the selection data at the time of printing — for example, beginning or ending date — you can include in the rules a variable which the program asks for at the time of printing.

When you want to print a single record on a invoice, for example, you design a print format. Once you have designed a print format, you can print a record right after you have entered it, in the “Filing” function. In our invoice example, you would enter the invoice, save the record, the print it out to send with a delivery. You set up rules to tell the program exactly where to print the data, each item more than once (if you wish), and your printout can be tailored exactly to the form you have. Comments to be printed on the form can be included as well.

The third way VersaForm prints a record is without using a print format — the program has the option of printing a copy of a record as it appears in the screen layout. You can make a complete printout of a record quickly for reference with this option.

OTHER FEATURES OF THE PROGRAM

The program comes on eight disks, and it’s often been recommended that putting it on a hard disk drive, like the Profile, is the only efficient way to use it. It’s true that VersaForm is very easily used on the Profile or similar hard disk. With the hard disk version, the programs load more quickly, and the user can use all 5 functions from one main menu. In the floppy disk version, each function is treated as a different program, and the user must reboot the computer (with control-reset) (with the boot disk and then the individual program disk) to use a different function. Although the floppy disk version may be awkward since the user must reboot when changing disks, the advantage of the floppy version is that you can use it to provide security of your data against unauthorized access.

For data entry, you can give someone the “Filing” disk and a data disk only, and they cannot print a report with those disks. Similarly, for printing reports, you can give someone the “Report” disk, a report work disk (blank) and a data disk only, and they cannot change the data with those disks. When you have the hard disk version, you can still use it as a floppy version.

All the program disks are completely copyable, and that means you can backup and restore your program disks quickly and without that $10 to $15 expense many companies charge on “locked programs”. The manuals are the best written manuals I’ve seen — one reference manual, one tutorial manual, and a hard disk installation manual. Company support has been excellent; upgrades are available at minimal cost ($20 to $30).

Overall performance of the program is quite good, although there are some minor irritating aspects of using it that can be tiresome. For instance, the program always asks you to enter the current date at the beginning of the “Filing” function, and the

beginning of the “Report” function. It would be convenient if it asked the current date, then remembered it or defaulted to that date when it next needed that information. Also, the “Report” function is pretty clumsy to use: you cannot abort the report once you start, and you have to go back to the menu just to print another report, which takes time. That’s also true of the “Filing” function: if you want to change files you have to go back to the menu.

THE FORM LAYOUT: SINGLE ITEMS AND REPEATING ITEMS

On the other hand, VersaForm offers a unusual feature which I haven’t seen in any other data base program. The form can be made up of both “Single” items and “Column” items. This is best illustrated by an example.

```
INVOICE

Inv # ........      Date ........

Sold To: ........................
         ........................
         ........................

Quantity    Stock #
Description      Price   Extension
____    ___     _____      __   __
__

                              TOTAL
$
        Thank you.
```

The data items such as Inv #, Date, Sold to, appear only once on each invoice. However, the data items such as Quantity, Stock#, Description, etc. are repeating items, and may appear more than once on each invoice. There may be only one occurrence of a repeating item, if only one stock item has been sold to this customer, or there may be 5 or 10 or 20 occurrences of a repeating item. VersaForm allows the user to have as many or as few entries in repeating items as each record needs. This has been a big problem with most data base programs on the market, which can handle mailing lists easily, but cannot satisfactorily handle an invoice with its variable length or repeating items. VersaForm is perfect for this application.

This model of single and repeating items adapts itself excellently to any number of applications — among them payroll records (print W-2s), client/medical records (print insurance claims and billing), and expense journals.

You’ll find that the program requires planning and a little time to set up your applications. The initial setup time varies with the complexity of your application; VersaForm suggests from between 1/2 hour to 2 hours, depending on the particular application. Because this could discourage the new user, VersaForm now has available sample application templates, similar to the templates available for VisiCalc and similar programs. These ready-made applications are the same forms you could set up yourself, but VersaForm designed them to shortcut setup time for the new user.

INTERFACE ROUTINES TO ACCESS THE DATABASE DATA

VersaForm also offers a set of Interface Routines, which are available separately. These routines are the key to accessing VersaForm data; a moderately experienced Pascal programmer can access all VersaForm files. You can write a program to read data from a text file into a VersaForm file, or you can read data from one VersaForm file into another VersaForm file. You can also print custom reports from VersaForm files, and do other applications involving VersaForm data. This is a very strong advantage for using VersaForm, because accessibility to data makes a “general” program easy to customize. Without this ability, you have to settle for just the report features which the original programmer wrote.

ADVANTAGES ON THE APPLE ///

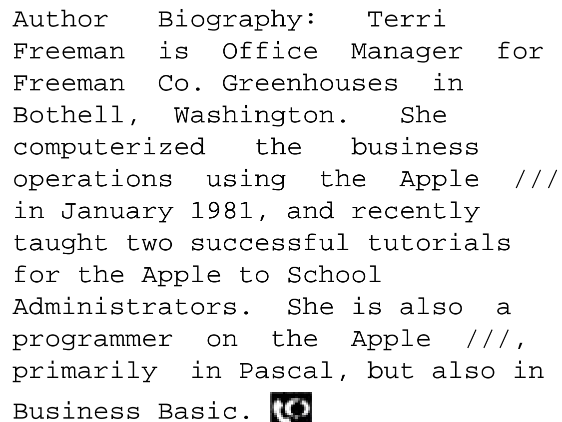
While VersaForm is available for many other microcomputers, such as the Apple II+ and IIe, the IBM PC, the DEC Rainbow, and some versions of the Atari and the TI, there are several nice features available when you use the Apple /// version. First, the Apple /// and the Profile work so well together, and VersaForm does work best when a hard disk is available. The Apple ///’s built-in numeric keypad offers convenient numeric data entry. Finally, the extra memory of the 256K Apple /// allows greater flexibility — you can create larger forms then on other machines, and there are added benefits when you use the Pascal Interface Routines.

The Apple /// is a powerful machine, and we’ve been waiting a long time for a program that matches the power and capability of this machine. This database is a versatile, yet user friendly program, and a superb member of the library of available Apple /// software.

VersaForm sells for $495; the Pascal Interface Routines sell for $249. Although the cost represents a substantial investment in software, it’s comparable to or lower than the cost of most dedicated accounting packages written for the Apple /// (or for the IBM PC, for that matter). And for the cost of this database, you can use it for all kinds of record keeping, which allows you to cut the cost per use. That’s not true for an dedicated accounting package.

Although the step-by-step tutorial provided by VersaForm is an excellent introduction to the program, next we’ll do a hands-on tutorial of sample applications for VersaForm. Finally, we’ll learn to use the VersaForm Pascal Interface routines. The uses of this program are limited, as the old phrase goes, only by your imagination!

/// /// ///

Author Biography: Terri Freeman is Office Manager for Freeman Co. Greenhouses in Bothell, Washington. She computerized the business operations using the Apple /// in January 1981, and recently taught two successful tutorials for the Apple to School Administrators. She is also a programmer on the Apple ///, primarily in Pascal, but also in Business Basic. 

Review: /// E-Z PIECES

Leon G. Stucki, Ph. D.

Product:

/// E-Z Pieces

From:

Haba Systems, Inc.
15154 Stagg Street
Van Nuys, California 91405

Synopsis:

This is clearly one of the most pleasant happenings in the Apple /// world for some time. Capturing the user friendly nature of Quickfile ///, improving both functionality and performance, and combining this with word processing and electronic spread sheets, Rupert Lissner has come up with a very handy "environment" for the Apple ///.

The three components (the Data Base, the Word Processor, the Spread Sheet) are combined with a simulated desktop that is obviously influenced by Lisa.

General Observations:

The bulk, if not all of the program, appears to have been coded in assembly language. Thus, the first observation that I had when comparing this system with Quickfile /// is that it is roughly an order of magnitude faster on the hard stuff. Sorting and rearranging within large data sets is very fast (roughly 6 to 10 times faster). Yet even more impressive is the speed improvement for "Finds".

Performing finds for large files in Quickfile /// is anything but quick. It is not uncommon to wait over a minute to retrieve the proper subset of records. It is also common to get ahead of the system when viewing the selected items. /// E-Z Pieces, on the other hand, performs these same tasks in just a couple of seconds.

The Desk Top

In an obvious attempt to create a Lisa-like feel, /// E-Z Pieces supports the Desk Top paradigm. This interface is very effective and pulls together a very powerful set of capabilities. One can, for instance, jump between numerous open files almost instantly. One can format diskettes at any time without losing text or time. And with the use of the "Cut and Paste" features it is very easy to copy and rearrange information rapidly.

The Data Base

The /// E-Z Pieces data base is a functional superset of Quickfile ///. Most of the commands are identical, but the performance has been greatly increased and the capacities of certain functions have also been increased.

The Word Processor

The /// E-Z Pieces Word Processor uses a Quickfile /// type command set. Sufficient

"Help" messages are provided and allow someone familiar with Quickfile /// to pick up the word processor almost immediately. Here again the speed is very good. Even using the fast repeat capabilities of the Apple ///, I was not able to get ahead of the machine. This is even true for fast scrolls. With Apple Writer /// it is quite easy to get ahead of yourself; this does not seem to be a problem with this system.

The Spread Sheet

The /// E-Z Pieces Spread Sheet also uses a Quickfile /// type syntax. It appears to be equivalent to the Advanced Version of VisiCalc. It supports very large data models and even allows columns of differing widths. The speed of recalculation seems slightly slower than the original VisiCalc but is fast enough to be very useful.

Problems and Observations:

The configurability for various printers is very valuable. However, I wasted quite a little while trying to install my PKASO parallel printer driver. I finally had to change the internal driver device type number from hex 40 to hex 41 before the printer configuration program would do what I wanted. This should be fixed. The Word Processor is almost a what-you-see-is-what-you-get program . . . but not quite. It's too bad that no one has yet attempted to use the graphics capabilities of the Apple /// to provide a more visual editor with various fonts. Having seen a prerelease copy of Draw-On /// at SoftCon, I am convinced the Apple /// is capable of much more than most people think.

Wish List:

This program wets the appetite in at least two ways. First, the concept of of the Desk Top should be expanded to support other user or commercial programs. In particular it would be very nice to be able to have a truly integrated graphics and chart-building capability. Mixed text and graphics may be further away, but it sure would be nifty if it could be combined with a truly graphic word processor.

/// Cheers Staff

Issue One
Dave Lingwood
Mike Christensen

Marlys Christensen
Terri Freeman
Bob Huelsdonk
Brian Matthews
Leon Stucki

About This Issue:

Originally dated Summer 1984, it was re-released in March 1985. Although The Apple /// was essentially dead as far as Apple was concerned, the dedicated people of A.P.P.L.E. wanted to give users something they could use for their computers in which they had invested so much time and money.

This dream died after the second issue of the magazine and went the way of so many other magazines in the 1980's.

/// CHEERS! AUTHOR'S GUIDE

/// CHEERS! enthusiastically and (well) cheerfully welcomes articles and programs from potential authors — we want to read stuff from other people!

We are interested in how-to-do-it articles, hardware/software reviews, use notes on popular applications software, and programs in any languages supported on the ///. Particular emphasis is given to programs that are instructive, and/or provide needed utilities.

The guidelines are rather simple: submit your text on DISK, along with source files for any programs, or source and invokable files for invokable modules. Disks should be in SOS or (gasp) DOS 3.3 or ProDOS formats. Articles, documentation and other "real text" may be in either ASCII or Pascal text files. Try not to depend on complex text formatting, unusual fonts, or graphics — remember that /// CHEERS! is designed for text output to both the screen and the wide variety of printers out there. We CAN accept articles and other text (not programs) in printed form, but note that page rate payments to authors are less in this case.

Follow the example of text in this edition regarding style and formatting. Text should NOT be right-justified.

We will acknowledge your submission by mail as soon as it is received. Rejected submissions will be returned to the author. We may also request revisions in text or programs. You will be notified by mail or phone if your submission is accepted, along with the approximate publication date.

You may request that programs be marked as "not for commercial use." Our normal practice and preference is that A.P.P.L.E. shall retain copyright to all published articles and programs. Other arrangements may be made, by discussion with the Editor.

Payment to authors is made at the rate of .72 cents per character for all published material (text and programs) submitted on disk. On-paper article submissions are paid at the rate of .5 cents per character. As noted above, programs must be submitted on disk.

Address all correspondence to:

Editor
/// CHEERS!
A.P.P.L.E.
21246 68th Ave. S.
Kent, WA 98032

or call: (206) 872-2245.

List Manager Source Listing

```
{
ListManager

WrittenbyD.MichaelChristensen
(C)1980,1984byDonovan’sReef
Apple///Pascal

Reference”ProgrammingInPascal”,Grogono,Addison-Wesley,1979
}

PROGRAMListMgr;
“USES
$Chainstuff;
“TYPE
$SetOfChar=SETOFChar;
“VAR
$CVal:String;

“PROCEDUREClearViewport;
“{DESCRIPTION
&ClearsviewportontheApple///}
$VAR
&PageCh:Char;
$BEGIN
&{Apple///clearscreen}
&PageCh:=Chr(28);
&Unitwrite(1,PageCh,1,0,12)
$END;{ClearViewport}

“FUNCTIONGetChar
“{PAR}
%(OkaySet:SetOfChar):Char;
“{DESC
&Getacharacterfromtheuser.
&ThecharactermustbeinthesetOkaySet.
&Ifitisnot,soundthebellandwaitforvalidcharacter.
&OkaySetdoesn’tneedtopassbothupperandlower—
&e.g.alowercase’a’isacceptedif’A’isintheOkaySet.}
$CONST
&ASCIIOffSet=32;
$VAR
&Good:Boolean;
&Ch:Char;
$BEGIN
&REPEAT
(Read(Keyboard,Ch);
({ApplePascalconvertscarriagereturnsintospaces,
*—convertbackintoChr(13)}
```

```
(IFEOLN(Keyboard)
*THENCh:=Chr(13);
(Good:=(ChINOkaySet);
({Ifuppercaseletterwastyped,acceptlowercasesubstitute}
(IF(NOTGood)
(AND(ChIN['A'..'Z'])
*THEN
,BEGIN
.{Converttolowercase}
.Ch:=Chr(Ord(Ch)+ASCIIOffSet);
.IF(ChINOkaySet)
0THENGood:=True
,END;{IF}
({Iflowercaseletterwastyped,acceptuppercasesubstitute}
(IF(NOTGood)
(AND(ChIN['a'..'z'])
*THEN
,BEGIN
.{Converttouppercase}
.Ch:=Chr(Ord(Ch)-ASCIIOffSet);
.IF(ChINOkaySet)
0THENGood:=True
,END;{IF}
(IFNOTGood
*THENWrite(Chr(7))
&UNTILGood;
&GetChar:=Ch
$END;{GetChar}
$
"FUNCTIONYes:Boolean;
$VAR
&Ch,Esc:Char;
$BEGIN
&Esc:=Chr(27);
&Ch:=GetChar(['Y','N','',Esc]);
&Yes:=(ChIN['Y',''])
$END;{Yes}

"PROCEDUREClearLine;
$VAR
&LineCh:Char;
$BEGIN
&LineCh:=Chr(30);
&Unitwrite(1,LineCh,1,0,12)
$END;{ClearLine}

"FUNCTIONPageCheck
"{PAR}
&(VARLine:Integer):Boolean;
$VAR
&Ch,Cr,Esc:Char;
&Continue:Boolean;
$BEGIN
&Cr:=Chr(13);
```

```
&Esc:=Chr(27);
&Continue:=True;
&IF(Line>20)
(THEN
*BEGIN
,Gotoxy(0,22);Write('PressSPACEtocontinue');
,Ch:=GetChar(['',Cr,Esc]);
,IF(ChIN['',Cr])THENClearViewport;
,Line:=2;
,Continue:=NOT(Ch=Esc)
*END;{IF}
&PageCheck:=Continue
$END;{PageCheck}
*
"PROCEDURERingList;
$TYPE
&HeapPointer=^Integer;
&DataPointer=^DataRecord;
&DataRecord=
(RECORD
*FrontPointer,BackPointer:DataPointer;
*Key:String;
(END;{Record}
&FileOfDataRecord=FILEOFDataRecord;
$VAR
&Base:DataPointer;
&Heap:HeapPointer;
&Data:DataRecord;
&DataFile:FileOfDataRecord;

${Ringoperations}
$
$PROCEDUREAscendingInsert
%(Base:DataPointer;
&Data:DataRecord);FORWARD;
$
$PROCEDUREDescendingInsert
%(Base:DataPointer;
&Data:DataRecord);FORWARD;
(
$FUNCTIONFindInRing
%(KeyToBeFound:String;
&Base:DataPointer;
&VARRef:DataPointer):Boolean;FORWARD;
`
$PROCEDUREInitRing
%(VARHeap:HeapPointer;
&VARBase:DataPointer);FORWARD;
$
$PROCEDUREInsertAfterInRing
%(Key:String;
&VARBase:DataPointer;
&Data:DataRecord;
&VARFound:Boolean);FORWARD;
```

```
(
$PROCEDUREInsertBeforeInRing
%(Key:String;
&VARBase:DataPointer;
&Data:DataRecord;
&VARFound:Boolean);FORWARD;
$
$PROCEDURESortAscending
&(VARBase:DataPointer);FORWARD;
&
$PROCEDURESortDescending
&(VARBase:DataPointer);FORWARD;
(
$PROCEDUREAppendRing
${PAR}
'(VARBase:DataPointer;
(Data:DataRecord);
&VAR
(NewPointer:DataPointer;
&BEGIN
(New(NewPointer);
({Addnewdata}
(NewPointer^:=Data;
(WITHNewPointer^DO
*BEGIN
,{Linkintoringatcurrentendofring}
,FrontPointer:=Base;
,BackPointer:=Base^.BackPointer;
,Base^.BackPointer^.FrontPointer:=NewPointer;
,Base^.BackPointer:=NewPointer
*END
&END;{AppendRing}

$PROCEDUREAscendingInsert;
${DESC
(Addalphabetically—insertarecordfromoldringintosortring.
(Caution:thisroutineiscasesensitive!}
&VAR
(Loop:DataPointer;
(Discard:Boolean;
&BEGIN
(Loop:=Base;
(REPEAT
*Loop:=Loop^.FrontPointer
(UNTIL(Loop=Base)OR(Data.Key<Loop^.Key);
(InsertBeforeInRing(Loop^.Key,Base,Data,Discard)
&END;{AscendingInsert}

$PROCEDUREClearRing
${PAR}
'(VARHeap:HeapPointer;
(VARBase:DataPointer);
&BEGIN
(Release(Heap);
```

```
(InitRing(Heap,Base)
&END;{ClearRing}

$FUNCTIONCountMembers
${PAR}
'(Base:DataPointer):Integer;
&VAR
(Count:Integer;
(Loop,Start:DataPointer;
&BEGIN
(Loop:=Base^.FrontPointer;
(Start:=Loop;
(Count:=0;
(REPEAT
*Loop:=Loop^.FrontPointer;
*Count:=Count+1;
(UNTIL(Loop=Start);
(CountMembers:=Count-1
&END;{CountMembers}

$PROCEDUREDeleteFromRing
${PAR}
'(Key:String;
(VARBase:DataPointer;
(VARFound:Boolean);
&VAR
(Ref,NewPointer:DataPointer;
&BEGIN
(Found:=FindInRing(Key,Base,Ref);
(IFFound
*THEN
,BEGIN
.IFBase^.BackPointer=Ref
0THENBase^.BackPointer:=Ref^.BackPointer;
.Ref^.FrontPointer^.BackPointer:=Ref^.BackPointer;
.Ref^.BackPointer^.FrontPointer:=Ref^.FrontPointer
,END
&END;{DeleteFromRing}

$PROCEDUREDescendingInsert;
${DESC
(Addalphabetically—insertarecordfromoldringintosortring.
(Caution:Thisroutineiscasesensitive!}
&VAR
(Loop:DataPointer;
(Discard:Boolean;
&BEGIN
(Loop:=Base;
(REPEAT
*Loop:=Loop^.FrontPointer
(UNTIL(Loop=Base)OR(Data.Key>Loop^.Key);
(InsertBeforeInRing(Loop^.Key,Base,Data,Discard)
&END;{DescendingInsert}
```

```
$PROCEDUREDispAscending
${PAR}
`(PortLine:Integer;Base:DataPointer);
&VAR
(TempHeap:HeapPointer;
(Loop,Temp:DataPointer;
(Continue:Boolean;
&BEGIN
(Mark(TempHeap);
(ClearViewport;
(Temp:=Base;
(SortAscending(Temp);
(Loop:=Temp^.FrontPointer;
(REPEAT
*Continue:=PageCheck(PortLine);
*IFContinueTHENBEGIN
,Gotoxy(0,PortLine);
,Write(Loop^.Key);
,Loop:=Loop^.FrontPointer;
,PortLine:=PortLine+1
*END{IF}
(UNTILLoop=Temp;
(Release(TempHeap)
&END;{DispAscending}

$PROCEDUREDispBackward
${PAR}
`(PortLine:Integer;
(Base:DataPointer);
&VAR
(Loop:DataPointer;
(Continue:Boolean;
&BEGIN
(ClearViewport;
(Loop:=Base^.BackPointer;
(REPEAT
*Continue:=PageCheck(PortLine);
*IFContinueTHENBEGIN
,Gotoxy(0,PortLine);
,Write(Loop^.Key);
,Loop:=Loop^.BackPointer;
,PortLine:=PortLine+1
*END;{IF}
(UNTIL(Loop=Base)ORNOT(Continue)
&END;{DispBackward}

$PROCEDUREDispDescending
${PAR}
`(PortLine:Integer;Base:DataPointer);
&VAR
(TempHeap:HeapPointer;
(Loop,Temp:DataPointer;
(Continue:Boolean;
&BEGIN
```

```
(Mark(TempHeap);
(ClearViewport;
(Temp:=Base;
(SortDescending(Temp);
(Loop:=Temp^.FrontPointer;
(REPEAT
*Continue:=PageCheck(PortLine);
*IFContinueTHENBEGIN
,Gotoxy(0,PortLine);
,Write(Loop^.Key);
,Loop:=Loop^.FrontPointer;
,PortLine:=PortLine+1
*END{IF}
(UNTIL(Loop=Temp)ORNOT(Continue);
(Release(TempHeap)
&END;{DispDescending}

$PROCEDUREDispForward
${PAR}
`(PortLine:Integer;
(Base:DataPointer);
&VAR
(Loop:DataPointer;
(Continue:Boolean;
&BEGIN
(ClearViewport;
(Loop:=Base^.FrontPointer;
(REPEAT
*Continue:=PageCheck(PortLine);
*IFContinueTHENBEGIN
,Gotoxy(0,PortLine);
,Write(Loop^.Key);
,Loop:=Loop^.FrontPointer;
,PortLine:=PortLine+1
*END{IF}
(UNTIL(Loop=Base)ORNOT(Continue)
&END;{DispForward}

$PROCEDUREExchangeInRing
${PAR}
`(Key:String;
(Base:DataPointer;
(NewData:DataRecord;
(VARFound:Boolean);
${DESC
(Tradedatainrecord}
&VAR
(Ref:DataPointer;
&BEGIN
(Found:=FindInRing(Key,Base,Ref);
(IFFound
*THEN
,BEGIN
.NewData.FrontPointer:=Ref^.FrontPointer;
```

```
.NewData.BackPointer:=Ref^.BackPointer;
.Ref^:=NewData
,END
&END;{ExchangeInRing}

$FUNCTIONFindInRing;
&VAR
(Loop,Start:DataPointer;
&BEGIN
(Loop:=Base^.FrontPointer;
(Start:=Loop;
(REPEAT
*Loop:=Loop^.FrontPointer;
(UNTIL(Loop^.Key=KeyToBeFound)OR(Loop=Start);
(IF(Loop=Start)AND(Loop^.Key<>KeyToBeFound)
*THENFindInRing:=False
*ELSE
,BEGIN
.Ref:=Loop;
.FindInRing:=True
,END
&END;{FindInRing}

$PROCEDUREInitRing;
&BEGIN
(Mark(Heap);
(New(Base);
(WITHBase^DO
*BEGIN
,FrontPointer:=Base;
,BackPointer:=Base;
,Key:='Thelistisnowempty.'
*END
&END;{InitRing}

$PROCEDUREInsertAfterInRing;
&VAR
(Ref,NewPointer:DataPointer;
&BEGIN
(Found:=FindInRing(Key,Base,Ref);
(IFFound
*THEN
,BEGIN
.New(NewPointer);
.NewPointer^:=Data;
.IFBase^.BackPointer=Ref
0THENBase^.BackPointer:=NewPointer;
.NewPointer^.FrontPointer:=Ref^.FrontPointer;
.NewPointer^.BackPointer:=Ref;
.Ref^.FrontPointer^.BackPointer:=NewPointer;
.Ref^.FrontPointer:=NewPointer;
,END
&END;{InsertAfterInRing}
```

```
$PROCEDUREInsertBeforeInRing;
&VAR
(Ref,NewPointer:DataPointer;
&BEGIN
(Found:=FindInRing(Key,Base,Ref);
(IFFound
*THEN
,BEGIN
.New(NewPointer);
.NewPointer^:=Data;
.NewPointer^.FrontPointer:=Ref;
.NewPointer^.BackPointer:=Ref^.BackPointer;
.Ref^.BackPointer^.FrontPointer:=NewPointer;
.Ref^.BackPointer:=NewPointer
,END
&END;{InsertBeforeInRing}

$FUNCTIONLengthOfLongestKey
${PAR}
`(Base:DataPointer):Integer;
&VAR
(Longest,LenBuffer:Integer;
(Loop,Start:DataPointer;
&BEGIN
(Loop:=Base^.FrontPointer;
(Start:=Loop;
(Longest:=0;
(REPEAT
*Loop:=Loop^.FrontPointer;
*LenBuffer:=Length(Loop^.Key);
*IF(LenBuffer>Longest)
*ANDNOT(Loop=Base)
,THENLongest:=LenBuffer
(UNTIL(Loop=Start);
(LengthOfLongestKey:=Longest
&END;{LengthOfLongestKey}

$PROCEDUREReadRing
${PAR}
`(VARDataFile:FileOfDataRecord;
(VARFilename:String;
(VARBase:DataPointer);
&VAR
(Data:DataRecord;
&BEGIN
(ClearRing(Heap,Base);
(Reset(DataFile,Filename);
(Data:=DataFile^;
(AppendRing(Base,Data);
(REPEAT
*Get(DataFile);
*Data:=DataFile^;
*IFNOTEOF(DataFile)
,THENAppendRing(Base,Data)
```

```
(UNTILEOF(DataFile);
(Close(DataFile)
&END;{ReadRing}

$PROCEDURESaveRing
${PAR}
'(VARDataFile:FileOfDataRecord;
(VARFilename:String;
(Base:DataPointer);
&VAR
(Loop:DataPointer;
&BEGIN
(Loop:=Base^.FrontPointer;
(Rewrite(DataFile,Filename);
(Get(DataFile);
(REPEAT
*DataFile^:=Loop^;
*Put(DataFile);
*Loop:=Loop^.FrontPointer
(UNTILLoop=Base;
(Close(DataFile,Lock)
&END;{SaveRing}

$PROCEDURESortAscending;
&VAR
(SortBase:DataPointer;
(SortHeap:HeapPointer;
(Loop:DataPointer;
&BEGIN
(InitRing(SortHeap,SortBase);
(Loop:=Base^.FrontPointer;
(REPEAT
*AscendingInsert(SortBase,Loop^);
*Loop:=Loop^.FrontPointer
(UNTIL(Loop=Base);
(Base:=SortBase
&END;{SortAscending}

$PROCEDURESortDescending;
&VAR
(SortBase:DataPointer;
(SortHeap:HeapPointer;
(Loop:DataPointer;
&BEGIN
(InitRing(SortHeap,SortBase);
(Loop:=Base^.FrontPointer;
(REPEAT
*DescendingInsert(SortBase,Loop^);
*Loop:=Loop^.FrontPointer
(UNTIL(Loop=Base);
(Base:=SortBase
&END;{SortDescending}
&
${Userinteractionroutines}
```

```
)
$PROCEDUREAskRing
${PAR}
`(VARDateFile:FileOfDateRecord;
(VARBase:DataPointer);
&VAR
(Leave:Boolean;
(Ch:Char;

&PROCEDUREAskRecord
&{PAR}
)(Title:String;
*VARData:DataRecord);
(BEGIN
*{Getdatahere}
*Gotoxy(0,0);
*ClearLine;
*Write(Title,':',Title,'whatname?');
*Readln(Data.Key);
*IFLength(Data.Key)>0
,THEN
.BEGIN
0{Removeanyleadingcontrolcodes}
0IFOrd(Data.Key[1])<=32
2THENData.Key:=''
.END{IF}
(END;{AskRecord}

&PROCEDUREAskAppendRing
&{PAR}
)(VARBase:DataPointer);
(VAR
*Data:DataRecord;
(BEGIN
*AskRecord('Add',Data);

*IF(Length(Data.Key)>0)
,THEN
.BEGIN
0{Linkintoring}
AppendRing(Base,Data);
{Displayresult}
DispForward(2,Base)
.END{IF}
(END;{AskAppendRing}

&PROCEDUREAskChange
&{PAR}
)(VARHeap:HeapPointer;
*VARBase:DataPointer);
(VAR
*Ch,Esc:Char;

(PROCEDURENotFound;
```

```
*VAR
,Ch,Cr,Esc:Char;
*BEGIN
,Cr:=Chr(13);
,Esc:=Chr(27);
,Gotoxy(0,0);
,ClearLine;
,Write(Chr(7),'Error:Recordwasnotfound.');
,Ch:=GetChar([Cr,Esc,''])
*END;{NotFound}

(PROCEDUREAskInsertInRing
({PAR}
+(VARBase:DataPointer);
*VAR
,Data:DataRecord;
,Ref:DataPointer;
,RefKey,NewKey:String;
,Before,Found:Boolean;
,Ch,Esc:Char;

*PROCEDUREAskAscendingInsert
*{PAR}
-(VARBase:DataPointer);
,VAR
.Data:DataRecord;
,BEGIN
.AskRecord('Ascendinginsert',Data);

.IF(Length(Data.Key)>0)
THEN
BEGIN
{Linkintoring}
AscendingInsert(Base,Data);

{Displayresult}
DispForward(2,Base)
END{IF}
,END;{AskAscendingInsert}

*PROCEDUREAskDescendingInsert
*{PAR}
-(VARBase:DataPointer);
,VAR
.Data:DataRecord;
,BEGIN
.AskRecord('Descendinginsert',Data);

.IF(Length(Data.Key)>0)
THEN
BEGIN
{Linkintoring}
DescendingInsert(Base,Data);
```

```
{Displayresult}
DispForward(2,Base)
END{IF}
,END;{AskDescendingInsert}

*PROCEDUREAskWhereToInsert
*{PAR}
-(VARRefKey:String);
,BEGIN
.Gotoxy(0,0);
.ClearLine;
.Write('Insert');
.IFBefore
THENWrite('before')
ELSEWrite('after');
.Write('what?');
.Readln(RefKey)
,END;{AskWhereToInsert}

*BEGIN{AskInsertInRing}
,Esc:=Chr(27);
,REPEAT
.Gotoxy(0,0);
.ClearLine;
.Write('Insert:BeforeFollowingAscendingDescendingQuit');
.Ch:=GetChar(['B','F','A','D','Q',Esc]);
.CASEChOF
'B':
BEGIN
Before:=True;
Gotoxy(0,0);
ClearLine;
AskWhereToInsert(RefKey);
AskRecord('Insert',Data);
IF(Length(Data.Key)>0)
THEN
BEGIN
:{Linkintoring}
:InsertBeforeInRing(RefKey,Base,Data,Found);
:IFFound
<THENDispForward(2,Base)
<ELSENotFound
END{IF}
END;
'F':
BEGIN
Before:=False;
Gotoxy(0,0);
ClearLine;
AskWhereToInsert(RefKey);
AskRecord('Insert:',Data);
IF(Length(Data.Key)>0)
THEN
BEGIN
```

```
:{Linkintoring}
:InsertAfterInRing(RefKey,Base,Data,Found);
:IFFound
<THENDispForward(2,Base)
<ELSENotFound
END{IF}
END;
'A':AskAscendingInsert(Base);
'D':AskDescendingInsert(Base)
.END{CASE}
,UNTIL(ChIN['Q',Esc])
*END;{AskInsertInRing}

(PROCEDUREAskDeleteFromRing
({PAR}
+(VARBase:DataPointer);
*VAR
,Key:String;
,Found:Boolean;
*BEGIN
,Gotoxy(0,0);
,ClearLine;
,Write('Delete:Nametobedeleted?');
,Readln(Key);
,DeleteFromRing(Key,Base,Found);
,IFFound
.THENDispForward(2,Base)
.ELSENotFound
*END;{AskDelete}

(PROCEDUREAskExchangeRecord
({PAR}
+(VARBase:DataPointer);
*VAR
,Key:String;
,Data:DataRecord;
,Found:Boolean;
*BEGIN
,{Getoldrecordkey}
,Gotoxy(0,0);
,ClearLine;
,Write('Exchange:Nametoreplaceinexchange?');
,Readln(Key);

,AskRecord('Exchange',Data);
,IF(Length(Data.Key)>0)
.THEN
BEGIN
{Linkintoring}
ExchangeInRing(Key,Base,Data,Found);
IFFound
THENDispForward(2,Base)
ELSENotFound
END{IF}
```

```
*END;{AskExchangeRecord}

(PROCEDUREAskSort
({PAR}
+(VARBase:DataPointer);
*VAR
,Ch,Esc:Char;
*BEGIN
,Esc:=Chr(27);
,REPEAT
.Gotoxy(0,0);
.ClearLine;
.Write('Sort:');
.Write('Ascending');
.Write('Descending');
.Write('Quit');
.Ch:=GetChar(['A','D','Q',Esc]);
.IFNOT(ChIN['Q',Esc])
THEN
BEGIN
CASEChOF
'A':SortAscending(Base);
'D':SortDescending(Base)
END;{CASE}
DispForward(2,Base)
END{IF}
,UNTIL(ChIN['Q',Esc])
*END;{AskSort}
(
(BEGIN{AskChange}
*Esc:=Chr(27);
*REPEAT
,Gotoxy(0,0);
,ClearLine;
,Write('Change:');
,Write('Clear');
,Write('Delete');
,Write('Exchange');
,Write('Insert');
,Write('Sort');
,Write('Quit');
,Ch:=GetChar(['C','D','E','I','S','Q',Esc]);
,IFNOT(ChIN['Q',Esc])
.THEN
BEGIN
CASEChOF
'C':
BEGIN
ClearRing(Heap,Base);
DispForward(2,Base)
END;
'D':AskDeleteFromRing(Base);
'E':AskExchangeRecord(Base);
'I':AskInsertInRing(Base);
```

```
'S':AskSort(Base)
END{CASE}
END{IF}
*UNTIL(ChIN['Q',Esc])
(END;{AskChange}
(
&PROCEDUREAskReadRing
&{PAR}
)(VARDataFile:FileOfDataRecord;
*VARBase:DataPointer);
(VAR
*Filename:String;
(BEGIN
*{Getfilename}
*Gotoxy(0,0);
*ClearLine;
*Write('Read:Getwhatfile?');
*Readln(Filename);

*{Retrieve}
*ReadRing(DataFile,Filename,Base);

*{Display}
*DispForward(2,Base)
(END;{AskReadRing}

&PROCEDUREAskSaveRing
&{PAR}
)(VARDataFile:FileOfDataRecord;
*Base:DataPointer);
(VAR
*Filename:String;
(BEGIN
*{Getfilename}
*Gotoxy(0,0);
*ClearLine;
*Write('Save:Keepunderwhatfilename?');
*Readln(Filename);

*{Savetodisk}
*SaveRing(DataFile,Filename,Base)
(END;{AskSaveRing}
&PROCEDUREAskView
&{PAR}
)(VARBase:DataPointer);
(VAR
*Ch,Esc:Char;

(PROCEDUREAskCount
({PAR}
+(VARBase:DataPointer);
*VAR
,Ch,Cr,Esc:Char;
,N:Integer;
```

```
*BEGIN
,Cr:=Chr(13);
,Esc:=Chr(27);
,N:=CountMembers(Base);
,Gotoxy(0,0);
,ClearLine;
,Write(‘Count:Thereare’,N,’entriesinlist’);
,Ch:=GetChar([Cr,Esc,’’])
*END;{AskCount}

(PROCEDUREAskLengthOfLongestKey
({PAR}
+(VARBase:DataPointer);
*VAR
,Ch,Cr,Esc:Char;
,N:Integer;
*BEGIN{AskLengthOfLongestKey}
,Cr:=Chr(13);
,Esc:=Chr(27);
,N:=LengthOfLongestKey(Base);
,Gotoxy(0,0);
,ClearLine;
,Write(‘Longest:’,N,’charactersisthelongest’);
,Ch:=GetChar([Cr,Esc,’’])
*END;{AskLengthOfLongestKey}

(BEGIN{AskView}
*Esc:=Chr(27);
*REPEAT
,Gotoxy(0,0);
,ClearLine;
,Write(‘View:’);
,Write(‘Foreward’);
,Write(‘Backward’);
,Write(‘Ascending’);
,Write(‘Descending’);
,Write(‘Count’);
,Write(‘Longest’);
,Write(‘Quit’);
,Ch:=GetChar([‘F’,’B’,’A’,’D’,’C’,’L’,’Q’,Esc]);
,IFNOT(ChIN[‘Q’,Esc])
.THEN
BEGIN
CASEChOF
‘F’:DispForward(2,Base);
‘B’:DispBackward(2,Base);
‘A’:DispAscending(2,Base);
‘D’:DispDescending(2,Base);
‘C’:AskCount(Base);
‘L’:AskLengthOfLongestKey(Base)
END{CASE}
END{IF}
*UNTIL(ChIN[‘Q’,Esc])
(END;{AskView}
```

```
&BEGIN{AskRing}
(Leave:=False;
(REPEAT
*Gotoxy(0,0);
*ClearLine;
*Write('List:');
*Write('Add');
*Write('Change');
*Write('Read');
*Write('Save');
*Write('View');
*Write('Quit');
*Ch:=GetChar(['A','C','R','S','V','Q']);
*CASEChOF
,'A':AskAppendRing(Base);
,'C':AskChange(Heap,Base);
,'R':AskReadRing(DataFile,Base);
,'S':AskSaveRing(DataFile,Base);
,'V':AskView(Base);
,'Q':
.BEGIN
Gotoxy(0,0);
ClearLine;
Write('Leave?<Y/N>');
Leave:=Yes
.END
*END{CASE}
(UNTILLeave
&END;{AskRing}

$BEGIN{RingList}
&InitRing(Heap,Base);
&DispForward(2,Base);
&AskRing(DataFile,Base)
$END;{RingList}

"BEGIN{ProgramListMgrMain}
$GetCVal(CVal);
$SetChain(CVal);
$ClearViewport;
$
$RingList
"END.

{(C)1980,1984byDonovan'sReef}
```

Call-A.P.P.L.E. In Depth CD-Rom

Four Great Books in one CD-Rom

All in PDF format and .DSK format, Software for Volumes 1 - 3 included on CD-Rom.

$24.95*