NO. 72

U.S. Edition: $2.50
International Edition: $3.00

JUNE 1984

MICRO™

for the Serious Computerist

In This Issue: 16 Pages of Apple IIe PEEKs, POKEs, CALLs

Better Random Number Generator

Musical Notes

16 Bit 68000 Supermicros

Programming with Macros

The Fast BASIC Compiler

A stunning show delighted the crowd at the Whisman Theater in Mountain View last night. Called *BLITZ!*, loaded and performed by Robert Skyles in a one-man virtuoso programming display, the show features the spectacular compiler for the COMMODORE 64.

"...BASIC programs running up to 20 times faster"

The *BLITZ!* compiler is faster than PET SPEED, and faster than any other Commodore compiler that has appeared to date. Shortly after Skyles took his seat and inserted *BLITZ!*, he had normal BASIC programs running up to 20 times faster after he *BLITZ*ed them.

The performer explained that *BLITZ!* translates the slow BASIC language into a much faster code, thus improving the performance of the BASIC routines. *BLITZ!* reads the entire BASIC program, decides which operations only have to run once, and compiles the operations. It then re-writes the program into its special P-code.

Skyles also showed how *BLITZ!* adds security to your programs, because once a program has been compiled, it is not readable. That means protection is an automatic part of the re-writing.

The highlight of the show was, for this reviewer, when *BLITZ!* compiled a string of BASIC programs such that one loaded the next. An impressed audience looked on as Skyles effortlessly passed information from one program to another.

BLITZ! on disk for the Commodore 64 costs only $99.00.
(You can also get one for the older PET CBMs on a special-order basis. It puts on quite a show!)

Skyles Electric Works
231E South Whisman Road
Mountain View, CA 94041
(415) 965-1735

Available from your local Commodore 64 dealer or call 1-800-227-9998.

This Month in MICRO

This is a very special month for MICRO. It's been redesigned to make it easier to read and easier to use. The listings are being typeset in clearer, larger print, and they are now proofed by computer for typos before being published.

The Staff believes that it has chosen features which will truly interest and excite serious computerists. However, to be certain of this, an extensive, in-depth Reader Survey has been included in this issue. The answers we receive will guide the future direction of MICRO's editorial material; be certain that your opinions, your desires, your likes and dislikes are considered. Return the Survey (with additional comments if you like) and make MICRO the magazine that you want it to be.

Featured This Month

Random Number Generator — Based on seven years of research, this is one of the best RNG's you will ever find. Whether you want it for software development, games, gambling, computer simulation, scientific experimentation, or any of its myriad other uses, you will discover that it is in an understandable form which you can easily use in your own programs.

Musical Notes — For the budding musician in each of us, a program that gives you control over a five octive range covering the entire treble, bass and alto clefs. With a 200 note table and rhythmic variations, this is far more than just a toy. It even offers the unusual option of changing notes if you don't quite get your masterpiece right the first time.

Programming with Macros — For the advanced computerist who writes in Assembly Language, Macros can be the key to more efficient, cleaner, more easily debugged programs. They are a powerful tool in knowledgeable hands.

Under the Commodore 64 ROM — Use the entire potential of your computer and free up your BASIC memory without sacrificing program messages. Here, at last, is a way to print messages to the screen (even full screens) using the 16K of RAM located under the BASIC and Kernal ROM chips.

Sixteen Bit 68000 Supermicros — The 68000 is thought by many to be the 6502 of the '80s, the future of microcomputing. To keep you aware of the latest trends, two seasoned computerists share their views and insights into this relatively new chip family. Their thoughts may influence the directions that your own hardware and software planning take.

Useful Math Functions — Save yourself time and mathematical aggrevation with this practical compilation of defined functions assembled into a very friendly program. Once entered, the math formulas are at your disposal as needed without the frustration of entering them again and again.

Apple IIe Guide and Atlas — A very special gift to our readers this month is the complete Apple IIe Supplement to our best selling book," What's Where in the Apple." This will bring earlier copies of the book up-to-date with the material included in the latest printing. For those who have not yet discovered the importance of this book for your programming efficiency, this will give you a chance to see the type of material available to you. (An order form may be found on the inside back cover, if you would like to own a complete copy.)

Question Mark — For those who enjoy a good mystery, our staff has come up with something that may pique your curiosity. Test your computing knowledge and find the answer.

Inside the CIA — No, we haven't gone political - just practical. In his ongoing 'Interface Clinic' series, Ralph Tenny examines a toggle mode of operation useful for output and input of multiple bytes of parallel data, and the advantages and methods of using the Shift Register. He also looks at ways to interface directly to a microprocessor bus without damaging the computer.

Spotlight — Acorn, a new computer system widely used in Great Britain, but just coming into American markets, is studied in detail. Developed for education, this versatile, sophisticated system with its excellent color graphics and advanced sound should go far in hobbyist, home and business applications.

editorial

Dear Readers,

As you read through this issue you will notice a few new things in Micro. While preserving the integrity and thrust of Micro, we are always working towards improving what we already have. To this end we have made some changes in the physical layout of the magazine to make reading Micro even easier and more enjoyable. You will notice that now we are typesetting our listings rather than taking them directly from the printer. This is in direct response to readers' comments on the legibility of listings. In addition to being typeset, the size of the actual type in the listings is slightly larger. (Hopefully this will help slow down the loss of your eyesight due to staring at too many computer screens for too many years.) We have also improved the layout of the articles to make reading easier.

Now some of you may feel that these changes reflect a loss of the 'original' Micro. To the contrary, we are more committed than ever to bring you articles that are intelligent and thought provoking. As part of this commitment we have added to some articles a 'Key to Understanding.' Don't get your hopes up; this is not some magical method to 'knowledge.' Nor is it a leftover from 'Secrets of the East.' Instead, it is our way of making more of our articles accessible to more readers. How often have you picked up a magazine and found that due to a lack of some assumed 'basics' an article was beyond your reach. If only you had a basic foundation you could then use the article. Or on the flip side, you come across an article which, although it has information you find interesting, is interspersed with Pablum explaining every other word. How many times have we read what a binary digit is? To help eliminate both of these problems we have taken out the basic information needed to understand an article and put it in a sidebox. This 'Key to Understanding' explains any terms or concepts that are necessary for intelligently reading the accompanying article. Those who are already familiar with the subject matter can go on to the article, being spared what for them would be repetitious. This tool will be used as is appropriate and necessary. In this issue you will find two articles that utilize this style - The Random Number Generator by Cem Kaner and John Vokey, and Programming with Macros by Patty Westerfield.

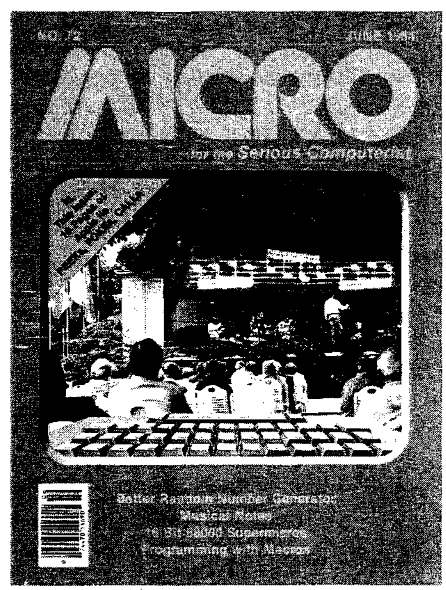
And now you have an opportunity to outdo yourselves -yes, it's Survey Time. (Why aren't you jumping up and down?) Last year Micro readers proved their stuff with a return rate of over 20 percent! In the world of surveys this is fantastic. Now you can do it again; don't miss out - this is your big chance to help out your fellow man (i.e. the Micro staff and Micro readers). And it is faster, easier, and much more pleasant than giving blood, although some have likened it to pulling teeth. Seriously, we would greatly appreciate your taking a few minutes of your time to fill out the survey and return it to us. We will pay the postage and put in the time and expense necessary to tabulate it. Why? It is through the survey results that we can decide how best to serve you. Everyone could be waiting for an article on interfacing your computer to your pet dog, but unless you fill out the survey and tell us we will never know. In the past, Micro readers have shown their stuff by responding in numbers much better than usually projected for survey returns. We hope this year to do even better. Although we can't give out a lollipop for each survey returned, we can guarantee that your opinion and information count and will be responded to.

In closing, I would like to reiterate that we feel we are here to serve you and not the other way aroun. Micro is not just a magazine, but rather a community of dedicated readers. We invite you to participate and come out and play - write a letter to the editor, submit articles, give us a call, or - if you find it's Friday night and your computer is down - fill out the survey. Thanks.

Mark S. Morano

Mark S. Morano
Technical Editor

On The Cover



Summer is here and music is in the air. Play the old favorites or compose a new tune to honor the season, with Musical Notes for the Apple.

COMPUTER MAIL ORDER

NEC PRINTERS
NEC 2050 ... $999.00
NEC 3550 ... $1679.00

PERCOM/TANDOM DISK DRIVES
5¼" 320K Floppy ... $229.00
5 Meg Hard w/Controller ... CALL
10 Meg Hard w/Controller ... CALL
15 Meg Hard w/Controller ... CALL
20 Meg Hard w/Controller ... CALL

AMDEK
310A Amber Monitor ... $169.00
DXY 100 Plotter ... $599.00
Color II ... $399.00

AST RESEARCH
Six Pak Plus...from ... $279.00
Combo Plus II...from ... $279.00
Mega Plus...from ... $309.00
I/O Plus...from ... $139.00

QUADRAM
Quadlink ... $479.00
Quadboard...as low as ... $289.00
Quad 512 Plus...as low as ... $249.00
Quadcolor...as low as ... $219.00
Chronograph ... $89.00
Parallel Interface Board ... $89.00
64K RAM Chips Kit ... $59.00

MICROPRO
WordStar/MailMerge ... $349.00
InfoStar ... $299.00
SpellStar ... $159.00
CalcStar ... $99.00

MICROSTUF
Crosstalk ... $105.00

MICROSOFT
Multiplan ... $159.00

ASHTON-TATE
dBASE II ... $389.00
Friday! ... $185.00

IUS
EasyWriter II ... $249.00
EasySpeller ... $119.00
EasyFiler ... $229.00

CONTINENTAL SOFTWARE
1st Class Mail/Form Letter ... $79.00
Home Accountant Plus ... $88.00

IBM

VISICORP

	IBM	APPLE
VisiCalc		$159.00
VisiCalc 4	$159.00	
VisiCalc Advanced		$269.00
VisiWord/Spell	$249.00	
Visitrend/Plot	$199.00	$199.00
VisiLink		$169.00
VisiFile	$199.00	$169.00
VisiSchedule	$199.00	$199.00
Visidex		$159.00
VisiPlot		$135.00
VisiTerm		$75.00
Desktop Plan	$199.00	$169.00
Bus.Forecast Model	$75.00	$75.00
StretchCalc	$75.00	$75.00
VisiTutor Calc	$59.00	$59.00
VisiTutor Advanced	$75.00	$75.00
VisiTutor Word	$259.00	$59.00
Visi—On Calc	$289.00	
Visi—On Graph	$179.00	
Visi—On Word	$275.00	
Visi—On Mouse	$159.00	
Visi—On Host	$319.00	

pfs

	IBM	APPLE
Write	$89.00	$79.00
Graph	$89.00	$79.00
Report	$79.00	$79.00
File	$89.00	$79.00
Solutions*: as low as	$16.00	$16.00

*Call On Titles

MONOGRAM
Dollars and Sense ... $109.00

LOTUS
1-2-3 ... $399.00

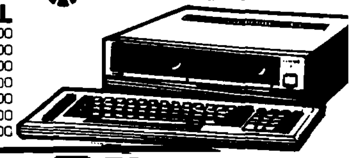
PROFESSIONAL SOFTWARE
PC Plus/The Boss ... $349.00

SYNAPSE
File Manager ... $89.00

SOFTWARE ARTS
TK Solver ... $189.00

MBC-550 ... CALL
MBC-555 ... CALL
MBC 1100 ... $1499.00
FDD 3200-320K Drive ... $389.00
MBC 1150 ... $1899.00
MBC 1200 ... $1849.00
FDD 6400-640K Drive ... $469.00
MBC 1250 ... $2099.00
PR 5500 Printer ... $599.00

SANYO



APPLE/FRANKLIN DISK DRIVES
MICRO-SCI
A2 ... $219.00
A40 ... $299.00
A70 ... $319.00
C2 Controller ... $79.00
C47 Controller ... $89.00

RANA
Elite 1 ... $279.00
Elite 2 ... $389.00
Elite 3 ... $569.00

APPLE IIe STARTER PACK
64K Apple IIe, Disk Drive & Controller, 80 Column Card, Monitor II & DOS 3.3
COMPLETE ... CALL

FRANKLIN

ACE 1000 Color Computer ... CALL
ACE Family Pack System ... CALL
ACE PRO PLUS System ... CALL
ACE 1200 Office Mgmt. System ... CALL
"Not IIe Expensive"

TERMINALS
914 ... $569.00
924 ... $689.00
925 ... $739.00
950 ... $929.00
970 ... $1039.00

COMPUTERS
Teleport Portable ... CALL
800A ... $1099.00
802 ... $2699.00
803 ... $1949.00
802H ... $4695.00
806/20 ... $4999.00
816/40 ... $9199.00
1602 ... $3399.00
1603 ... CALL

TeleVideo

PRINTERS

AXIOM
AT-100 Atari Interface ... $239.00
GP-100 Parallel Interface ... $199.00
AT-846 Interface ... $89.00

BMC
401 Letter Quality ... $589.00
BX-80 Dot Matrix ... $269.00

CENTRONICS
122 Parallel ... $299.00
739-1 Parallel ... $199.00
739-3 Serial ... $249.00

C.ITOH
Gorilla Banana ... $209.00
Prowriter 8510P ... $379.00
Prowriter 1550P ... $599.00
A10 (18 cps) ... $569.00
8600 P ... $929.00
F10-40 ... $999.00
F10-55 ... $1499.00

COMREX
ComWriter II Letter Quality ... $499.00

DIABLO
620 Letter Quality ... $949.00
630 Letter Quality ... $1749.00

DAISYWRITER
2000 ... $999.00
Tractor Feed ... $109.00

EPSON
MX-80FT, MX-100, RX-80, RX-80FT, FX-80, FX-100 ... CALL

IDS
Prism 80...For Configurations ... CALL
Prism 32...For Configurations ... CALL

MANNESMAN TALLY
160L ... $589.00
180L ... $799.00
Spirit 80 ... $309.00

NEC
2010/2030 ... $899.00
8023 Dot Matrix ... $379.00
8025 Dot Matrix ... $669.00
3510 Serial/Letter Quality ... $1449.00
3530 Parallel/Letter Quality ... $1499.00
7710/7730 Serial/Parallel ... $1949.00

OKIDATA
82, 83, 84, 92, 93, 2350, 2410 ... CALL

SMITH CORONA
TP-2 ... $399.00
Tractor Feed ... $119.00

SILVER REED
500 Letter Quality ... $469.00
550 Letter Quality ... $699.00

STAR
Gemini 10X ... $299.00
Gemini P15X ... $399.00
Delta 10 ... $559.00
Serial Board ... $75.00

TOSHIBA
1350 ... $1699.00

TRANSTAR
120P ... $499.00
130P ... $689.00
315 Color ... $499.00

APPLE INTERFACE CARDS & BUFFERS
Choose from PKASO, Orange Micro, MPC, MicroMax, Tymac, Quadram & Practical Peripherals. **PRINTER CABLES** are available for most all computers on the market. We supply all your computer needs.

PAPER SUPPLIES
1000 shts. 8½x11 Tractor Paper ... $19.99
1000 shts. 14½x11 Tractor Paper ... $24.99
1 or 2" Address Labels ... $9.99

MODEMS

ANCHOR
Volksmodem ... $59.00
Mark IL ... $79.00
Mark VII (Auto Ans./Auto Dial) ... $119.00
Mark XII (1200 Baud) ... $299.00
TRS-80 Color Computer ... $99.00
9 Volt Power Supply ... $9.00

HAYES
Smartmodem 300 ... $209.00
Smartmodem 1200 ... $499.00
Smartmodem 1200B ... $449.00
Micromodem IIe ... $269.00
Micromodem 100 ... $299.00
Smart Com II ... $89.00
Chronograph ... $199.00

NOVATION
J-Cat ... $99.99
SmartCat 103 ... $179.00
SmartCat 103/212 ... $399.00
AutoCat ... $219.00
212 AutoCat ... $549.00
Apple Cat II ... $249.00
212 Apple Cat ... $569.00
Apple Cat 212 Upgrade ... $309.00
Cat ... $139.99

ZENITH
ZT-1 ... $339.00
ZT-10 ... $309.00
ZT-11 ... $369.00

MONITORS

AMDEK
300 Green ... $149.00
300 Amber ... $159.00
310 Amber ... $169.00
Color 1 ... $279.00
Color 1 Plus ... $299.00
Color 2 ... $399.00
Color 2 Plus ... $419.00
Color 3 ... $349.00
Color 4 ... $699.00

BMC
12" Green ... $88.99
12" Green Hi-Res. ... $119.99
9191- 13" Color ... $249.00

GORILLA
12" Green ... $86.99
12" Amber ... $95.99

NEC
JB 1260 Green ... $109.00
JB 1201 Green ... $149.99
JB 1205 Amber ... $159.99
JC 1215 Color ... $269.00
JC 1216 RGB ... $429.00
JC 1460 Color ... $359.00

PRINCETON GRAPHICS
HX-12 RGB ... $519.00

SAKATA
100 ... $269.00

TAXAN
210 Color RGB ... $299.00
400 Med-Res RGB ... $319.00
415 Hi-Res RGB ... $439.00
420 Hi-Res RGB (IBM) ... $489.00
100 12" Green ... $125.00
105 12" Amber ... $135.00

USI
Pi 1, 9" Green ... $99.99
Pi 2, 12" Green ... $119.99
Pi 3, 12" Amber ... $149.99
Pi 4, 9" Amber ... $139.99
1400 Color ... $269.99

QUADRAM
Quadchrome 8400 ... $549.00

ZENITH
ZVM 122 Amber ... $109.00
ZVM 123 Green ... $89.99
ZVM 135 Color/RGB ... $469.99

east
800-233-8950
In PA call (717)327-9575, Dept. 40515
Order Status Number: 327-9576
Customer Service Number: 327-1450
477 E. 3rd St., Williamsport, PA 17701

canada
Ontario/Quebec
800-268-3974
Other Provinces 800-268-4559
In Toronto call (416)828-0866, Dept. 40515
Order Status Number: 828-0866
2505 Dunwin Drive, Unit 3B
Mississauga, Ontario, Canada L5L1T1

west
800-648-3311
In NV call (702)588-5654, Dept. 40515
Order Status Number: 588-5654
P.O.Box 6689
Stateline, NV 89449

MasterCard

No risk, no deposit on C.O.D. orders and no waiting period for certified checks or money orders. Add 3% (minimum $5) shipping and handling on all orders. Larger shipments may require additional charges. NV and PA residents add sales tax. All items subject to availability and price change. Call today for our catalog.

MICRO™

for the Serious Computerist

JUNE 1984

MICRO is published monthly by:
MICRO, Chelmsford, MA 01824
Second Class postage paid at:
Chelmsford, MA 01824 and additional mailing offices.
USPS Publication Number: 483470
ISSN: 0271-9002.
Send subscriptions, change of address, USPS Form 3579, requests for back issues and all othe fulfillment questions to:
MICRO
P.O. Box 6502
Chelmsford, MA 01824
or call 617/256-3649.
Subscription Rates: (per year):
U.S. $24.00 or $42.00 for two years
Foreign surface mail: $27.00
Air mail: Europe $42.00
Mexico, Central America, Middle East, North Africa, Central Africa $48.00
South America, South Africa, Far East, Australia, New Zealand $72.00

NO. 72

Product Reviews

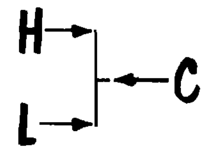
Departments

feedback

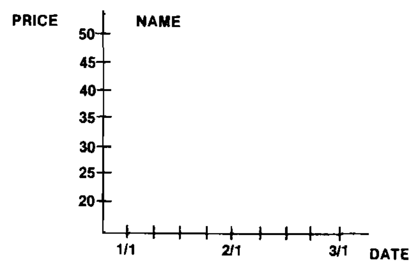
Dear Editor:

My programming abilities are just enough to get me into trouble. But I've been following your series on graphics and hope you can help me.

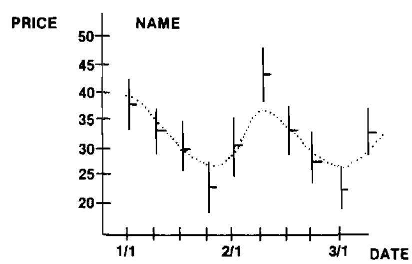
I want to graph a time series of data as line graphs. Data is: High, low, close, date and I want to display it in the form:



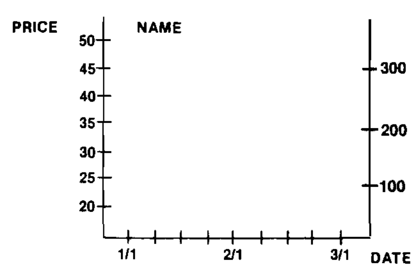
I also want to label the axis as to price and time:



Then overlay a moving average of the data:



Possibly adding a second series of data on the same chart, requiring a third label on axis:



The kicker is that data may cover an extended period of time (e.g. 200 days) and for clarity maybe only 50 days could be displayed at a time. So, I want to be able to scroll back and forth, timewise (left, right) and change the text labels as this occurs, stopping as necessary and then dumping the screen to a printer.

Big order? That's why I need help.

Harvey L. Taback
Vancouver B.C., Canada

68000: The 6502 of the '80's

Dear Editor:

It's a real pleasure to be writing to you using the Amdek monitor that you and your staff awarded me for "Country 5." I was really quite elated the day I received the registered letter announcing my good fortune in the Micro graphics contest. Thank you all very much for the recognition.

I've been a follower of Micro since the days of the KIM-1 computer which served as my training wheels in the world of 6502 programming. In fact, I still have the motherboard and proto board from the Computerist holding the ol' KIM system together.

As a reader of Micro I'd like to take this opportunity to make a suggestion that I believe will benefit many present Micro followers and perhaps attract a whole new following.

The 6502 obviously has a lot of life left in it. Apple has just introduced the Apple IIc and, as you must already know, the Western Design Center in Mesa, AZ is about to release the first full implementation of its 16 bit versions of the 6502 (65802 and 65816). That's great for all of us die hard 6502 programmers. I understand that Apple and Atari have already ordered a significant number of these chips for evaluation.

I believe, however, that the Motorola 68000 series of microprocessors will become the 6502 of the 80's. I know that you folks are already 6809 enthusiasts, so I don't expect to run into too much resistance to the idea of supporting another great Motorola product. In fact, I seem to recall a 68000 series of articles around the end of 1982. What happened?

I've begun programming the 68000 using the QPak-68 coprocessor board for the Apple II. The QPak-68 is a complete 68000 development package from QWERTY, Inc. It's based on the

68008 and is a superb product. Anyway, my initial reaction to the 68000 has been nothing if not enthusiastic. It's almost like working in a high level language after so many years of being zero page bound and, indeed, 8 bit bound with the 6502.

As you know, Apple has adopted the 68000 family of processors and Sinclair is about to unleash a $500 computer based on the 68008. There is no magazine that I know of that is supporting the 68000 as of yet. Why not do the world a favor and be the first to offer your readers a pathway into the current generation of high performance microprocessors.

I must sound like a member of the Motorola marketing team after that last paragraph. No, in fact I'm a relative newcomer to the 68000, but I see a vast future for this chip family and apparently an increasing number of computer systems designers do also. How about putting the question to your readers and find out how they feel about Micro supporting the 68000.

Once again thank you for the wonderful validation in selecting my Apple graphic as the first prize entry in the Apple II category.

Thomas Wilson
San Rafael, CA

Editor's Note: The staff of Micro also feels that the 68000 chip may well be the 6502 of the '80's. We need to know our readers' interest in a regular 68000 column and feature articles on this family from time to time. These would be in addition to (not in place of) our other chips. Please take a few minutes to answer the Reader Survey Questions on the card in this issue; we will analyze your responses carefully to determine the direction you want Micro to follow.

Medical Programs

Dear Editor:

Several months ago I wrote to you asking if any of your readers would be interested in contributing programs to a book, "*Microcomputer Programs In Medicine.*" The response from your readers was astounding.

I had letters, post cards, packages of discs and printouts from all over America, various parts of Canada, England, Ireland, South Africa, Saudi Arabia, Israel, Australia, Malaysia and even one from mainland China.

I had phone calls in the middle of the night from foreign parts apologizing for the time zone difference, but asking for details of the impending book.

As a result, the programs have now been published in book form in two volumes. Volume I contains scheduling and appointment programs, direct patient billing and accounts receivable, patient file retrieval, simple statistics including standard deviations, etc., graph drawing and curve fitting, numeric and alphabetic sorting. Volume II contains programs on patient history taking and history summarization, respiratory function, pediatric growth percentile calculation, bar graph drawing, analysis table making, using a VisiCalc template, obesity advisory program for weight loss, CHI square statistics and analysis of variance.

The book is now in print and is available from the publishers, Computer Medica Corporation, Medical Software Company, 328 Main Street, Center Moriches, N.Y. 11934, at $80 per volume.

I must thank your readers again for the fantastic response.

Derek Enlander, M.D.
New York, NY

MICRO

Don't Forget to Send Your Reader Survey. Do It Today!

Acorn Microcomputer System

Distributor

Acorn Computers Corporation
400 Unicorn Park Drive
Woburn, MA 01801

Introduction

The Acorn microcomputer was first developed in response to an invitation issued by the BBC to computer firms to compete in creating a new micro that would meet their specifications. The contract was awarded to Acorn, which, at the time, was only five years old.

Various features, in particular a Local Area Networking capability of up to 254 Acorns, led to the acceptance of the Acorn as an educational tool. Presently more than 85% of English schools use the Acorn. The Acorn has made its entrance into the U.S. market with a few model systems established, the most recently publicized being the school system of Lowell, Massachusetts, where a network of Acorns is serving grades K-12.

Memory and Optional Expansion Features

The Acorn has a series of co-processors that allow optional expansion of the standard 64K of memory. The Operating System is 16K built-in ROM, 16K built-in Word Processor (VIEW), built-in ROM BASIC interpreter, 32K RAM for User Programs. The co-processors enable the addition of three expansion features:
1) a 3MHz 6502 (includes an additional 64K RAM): this will run any program faster with more space available to the user
2) a Z-80B with 64K RAM: 'the software with this unit allows CP/M programs to be run with more memory than a normal CP/M environment. In addition, the main user program is left free to do calculations, leaving the BBC Microcomputer to deal with graphics, printers, clock, floppy disk, etc.'
3) a NS 16032: a 16 bit machine with 32 bit internal architecture, can be used with up to 16 Megabytes of RAM.

The Acorn has a built-in (ROM) BASIC interpreter, which also includes a 6502 Assembler. This permits Assembly Language to be mixed in the middle of a BASIC program. All the standard features and statements are available with some nice enhansions such as local variables, subroutines that pass parameters and recursion.

Also built into ROM is a 16K word processor called VIEW. This package is of professional quality featuring local and global control, search, change, replace, automatic page numbering, etc.

Graphics

When first viewing the Acorn one immediately notices the high quality graphics; an RGB Video is used to display the high resolution screens. The Acorn uses a number of display modes, including 640 x 200 for 2 color graphics (80 x 25 text), 320 x 200 for 4 color (40 x 25 text), and 160 x 200 for 16 color graphics (20 x 25 text), to list a few. There are a number of commands which facilitate graphics control, including the familiar commands such as PLOT, DRAW, and MOVE.

Sound/Music

To generate sound and music the Acorn employs four 'channels.' Through the use of SOUND and ENVELOPE commands a great deal of control is available to the user, and a full five octave range gives plenty of room to work in. The ENVELOPE offers a great deal of control with six parameters, governing the attack, decay, and release of a note.

Voice Synthesis

The Acorn also has a built-in voice synthesizer, including a Speech Processor and a PHROM (Phrase Read Only Memory). The Speech Processor is one made by Texas Instruments, the TMS 5220. In the PHROM chip is stored 206 ready-made words, and other PHROMs fitted with different words will be available in the future. The speech system can be accessed from BASIC and Assembly language.

Interfaces

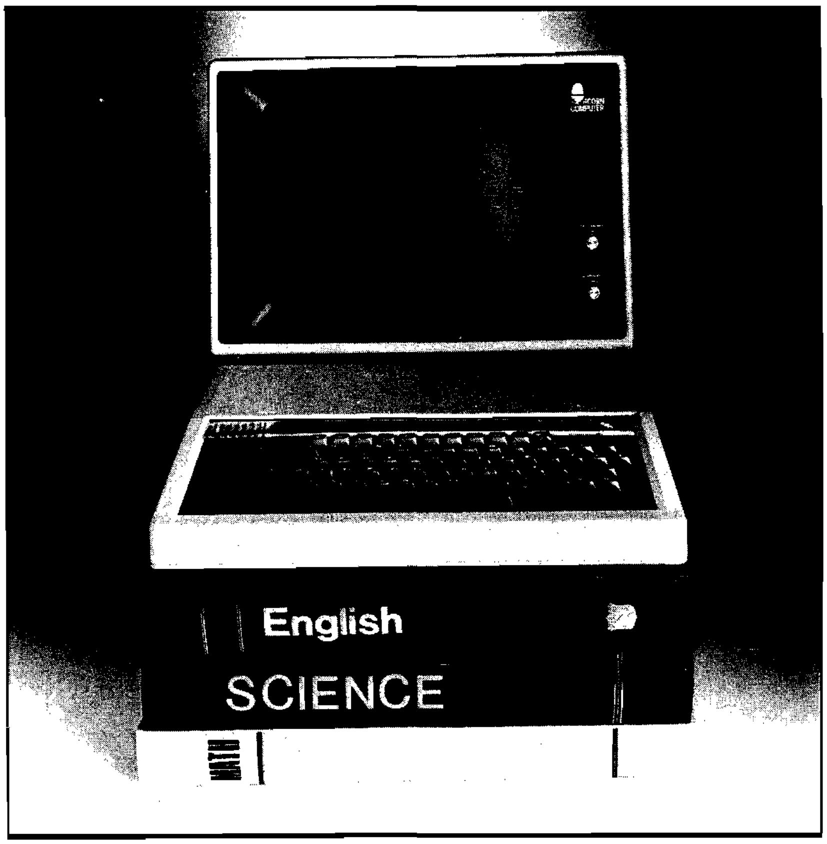
The Acorn includes a number of interfaces: Floppy Disk Interface up to 1 MB unformatted; RS423 Serial Interface (RS232 enhanced for speed and distance); Software Selectable Baud Rates between 75 and 19,200 Baud; 8-bit 'Centronics-type' parallel printer port; four 12-bit Analog Input Channels -input voltage range 0-1.8V, 10ms conversion time for each channel; standard audio cassette for low-cost storage.

Peripherals

Peripherals supported include: 5 1/4-inch floppy disk drives with capacities of 400K-800K formatted; monochrome, color (RGB, Composite Video) and TV; dot matrix and daisy-wheel printers, game paddles and joysticks.

Keyboard and Physical Description

The 73-key Qwerty keyboard is cleanly laid out, including 10 User Definable Function Keys. It has a nice touch and has the break key safely put out of normal reach. The size is 16''W x 13 1/2''D x 2 3/4''H, weighing in at 16 lbs. The dual disk drive is compact and neatly designed, taking the space of a normal-sized single disk drive.



Software

The software available for the Acorn is growing every day. Although all of the software that is presently in use in England isn't available here, there is certainly enough to keep anyone busy. There are packages covering business applications, graphics, languages and a plethora of educational software. American companies have been enlisted in converting some of the English software, in particular the education packages, for use in the United States. The name that stands out in this area is Krell Software Corporation (1320 Stony Brook Road, Stony Brook, NY 11790). The most well known of their software are Alexander the Great and Plato's Cave. Alexander the Great is a cross between Risk and Scrabble, developing word and arithmetic skills as well as touching upon ancient geography. The only problem is that it is fun and addictive. Plato's Cave is an introduction to the relation between evidence and inference (using a Platonian approach). The subjects covered by Krell and other manufacturers of Educational Software is quite varied, developed for all levels and covering subjects from color to transpiration to gas chromatography.

Price

The price breakdown is as follows: for the basic microcomputer the price is $995.00; the 800KB dual disk drive is $995; a 400KB (double sided) single disk drive is $545; a 200KB (single sided) single disk drive is $395; a RGB high resolution monitor (12 inch) is $595; Monochrome monitor (12 inch) amber or green is $195, both the RGB and Monochrome include cables. Prices for the additional co-processors are not available at this time.

Conclusion

Although the past emphasis has been in the area of education, the Acorn has just begun to conquer the many fields that it is capable of handling. Given its memory, telecommunication, graphic and other well developed features it certanly merits consideration for home or business use.

MICRO

Product Name: **Autoterm**
Equip. Req'd: Color Computer with 32K
Price: $39.95 cassette;$49.95 disk
Manufacturer: PXE Computing
11 Vicksburg Lane
Richardson, TX 75080

Description: A full-feature communication package with added features. An extensive amount of effort has been spent to insure user-friendliness; several detailed menus guide the operator in setup and operation. Any operation can be temporarily suspended or allowed to run in background mode while the user accesses a HELP screen. The communications ability seems to be standard - 110 to 1200 baud full- or half-duplex, with send and receive capability for text, graphics, BASIC and Assembly Language data. Full communications using any modem can continue in the background mode while data is reviewed or edited. The connection will not be broken during cassette loads and save, if you desire. Provisions for embedded text and menu-selected print options make it easy to use any printer. Received data can be printed in any menu-defined format, regardless of the width of text lines received.

An outstanding feature of this package is called keystroke multipliers. The purpose is to automate the sign-on procedures for various modems, i.e., invoke a keystroke multiplier which will make the connection, complete the contact and sign off, all automatically.

Pluses: The low cost of this software makes it viable for an unlimited number of simple control and measurement tasks, aside from its intended communications and editing ability. Although full utilization of the package would be complex, the learning process seems to be optimized and friendly.

Minuses: So far, no bugs have been found, and any perceived problem has been overcome with more study and experimentation.

Documentation: An 81-page manual details the operation of the program in a well written format, with additional reinforcement from the program itself. The book is well organized with a complete and logical index, and numerous detailed examples are used where needed.

Skill level: By the time a CoCo owner has progressed to the need or desire for communications, he will be ready to use this program.

Reviewer: Ralph Tenny

Product Name: **SuperText Professional**
Equip. Req'd: Apple II, II+, IIe with Applesoft ROM, DOS 3.3, 48K, lower case capability
Price: $175 ($99 special)
Manufacturer: MUSE Software, Inc.
347 N. Charles Street
Baltimore, MD 21201

Description: The most recent version of one of the first powerful Apple word procesors. With it, a skilled user can write, edit, store, preview, and print documents in a wide variety of formats. The program supports the Smarterm, Full-View 80, and Videx 80-column boards for the II+, and either of the IIe 80-column cards, as well as the Apple 40-column format. It is simple to configure the program for most of the popular printers.

Pluses: One of the most unusual features is the "Math Mode", which permits calculations within files. This is particularly useful for preparing invoices, cost estimates, and proposals. The screen can be split and each half scrolled and edited separately. A key can be defined as any string up to 30 characters long, useful for reviews where a single title occurs over and over. Cursor movement is smooth and unobtrusive. It seems to be nearly impossible to make such a serious error that text is lost from memory. You can easily set up multi-line running heads or feet, embed codes for bold, italic, and other tyepface changes, and save or load files. There is a quick reference card.

Minuses: SuperText creates nonstandard disk files. The program uses several of the same code sequences in different modes, and it is fairly easy to forget what mode is on. There is no provision for footnotes, super- or subscripts, or hyphenation.

The Apple IIe uses CTRL I to tab; SuperText has not provided a substitute control code to turn on italics printing, so it is necessary to embed a dummy character while entering text, then use the 'change' mode to alter it.

Documentation: The manual has no index and needs one. It is comprehensive, however, and almost any answer can be puzzled out by working through the extensive table of contents.

Skill level: It requires either experience with word processors or great persistence to learn. A person who learns the program and uses it regularly, however, will have the use of an effective writing tool.

Reviewer: K.C. Tinkel

Subscribe to MICRO...

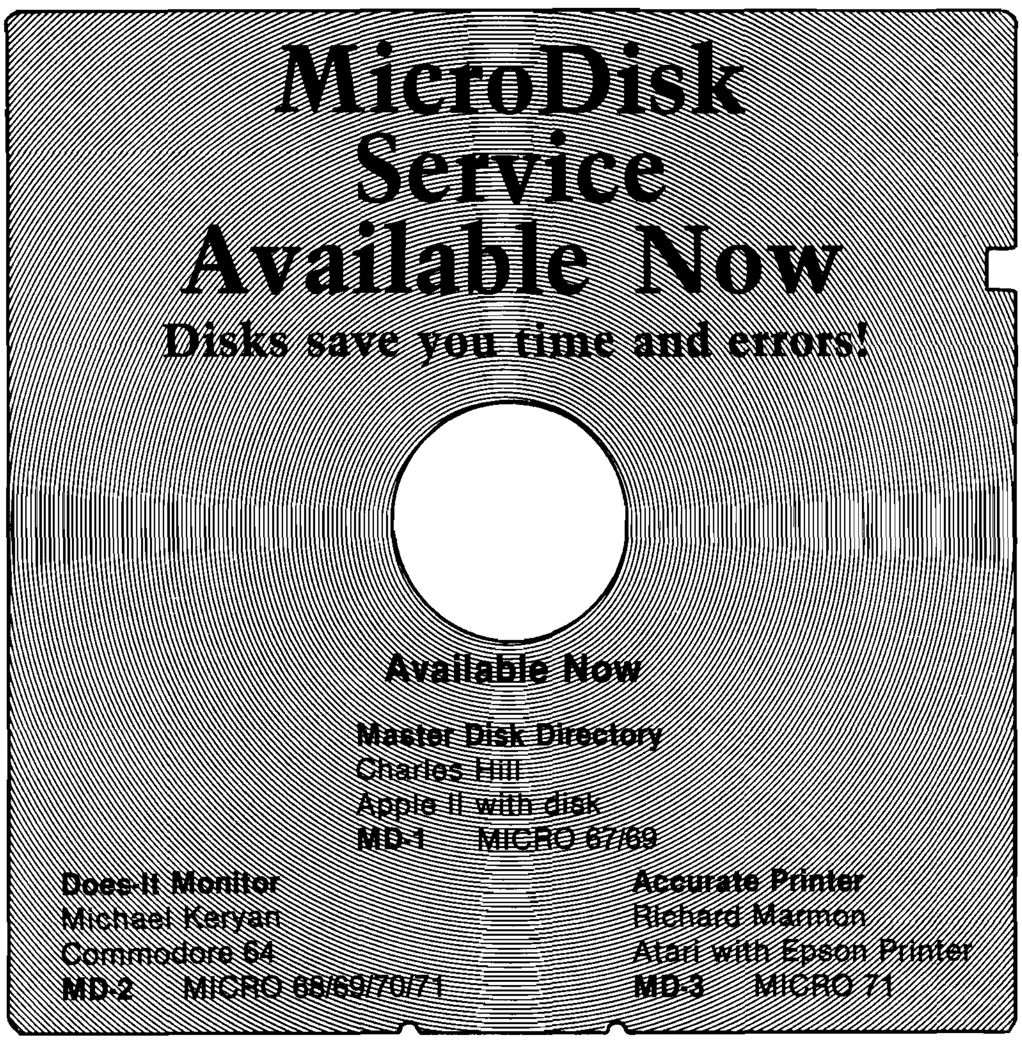
Save 20% and we'll send you a BONUS GIFT

with your subscription!

Receive a coupon good for one MicroDisk of your choice.
A $15.00 Value — FREE!

Each Disk contains a variety of programs from Micro, all entered and ready to run on your machine. Saves time, avoids errors.

Fill out the attached card and mail today!



Each diskette includes all of the programs in BASIC and/or Assembly Source, plus binary 'load-and-go' files. The price of only $15.00 includes shipping and handling.

Product Name: **Super-Text Professional Word Processor**
Equip. Req'd: Atari 400/800/1200XL, with minimum 48K
Price: $99.00
Manufacturer: Muse Software
347 N. Charles Street
Baltimore, MD 21201

Description: 'Super-Text Professional is designed to be a business-powered processor simple enough for home and educational use,' according to the developer. It contains Atari DOS making all DOS functions available to the user. All of the basics are included; delete, find/replace, block operations, cursor movement, local and global control.

Pluses: Starting with an Introduction and Help Menu, the user has a variety of choices and options available. The user can set parameters for his printer with most of the major printers parameters provided, the users simply selects the one he needs. The printer can also be controlled from within the text. Other nice features are automatic page numbering, single key commands (underlining with one command), format and tab specification control. Super-Text has a system status line displayed upon request which gives pertinent information when needed. Muse Software has also provided something called Autolink (trademark) which 'greatly increases the Atari's file organization and manipulation capabilities.' With this feature you can link files on the same or different disks and then do global finds, replaces, etc. through those linked files. There is a user defined function key called The Key whose character set you can define - up to thirty characters.

Minuses: Super-Text has seperate modes for Changes, Adds, and file manipulation. Changing back and forth between modes is a little awkward to start with. It is not at all like other word processors in this respect. To those who are familiar with other packages this method will undoubtedly seem a bit cumbersome at first. Once this peculiarity is gotten used to it becomes acceptable. Again it differs from other word processors in its use and definition of inserts. If you go searching for Insert instructions you will find it very frustrating. There isn't any defined Insert; rather through manipulation of the delete command and the Change and Add mode you can achieve what is an insert. For those not used to other packages I suspect neither the modes or insert would be a problem. Those who are familiar with other WP packages will find a period of adaptation to these different features is necessary.

Documentation: The manual provided with Super-Text is clearly written, with good chapter outlines. Unfortunately, as with many software packages, there is a continuation of the belief that indexes are obsolete.

Skill level: This package is geared more for the advanced WP user, having all of the advanced features such a user would want and use. Beginners would certainly be able to use Super-Text, and actually may benefit from the concept of modes, to seperate the various functions.

Reviewer: Mark S. Morano

Product Name: **G.A.L.E.**
Equip. Req'd: Apple II
Price: 49.95
Manufacturer: MicroSPARC, Inc.
10 Lewis St.
Lincoln, MA 01773

Description: A Global Applesoft Line Editor with edit mode, macro mode, global commands, hex/dec conversion, auto line number and help are easily accessed from BASIC or the monitor. It includes search and change, BLOAD information, free sectors, macro definitions for single key entry, a "hide" command to temporarily store a program, line finding, pointer dump, renumber, variable cross reference, append, converting hex to dec and vice-versa, and a line editor with insert, delete, lower case entry, find, verbatim entry, and a help screen.

Pluses: GALE is easy to use and is a great time saver for Applesoft programmers. It includes the most complete set of commands among the current popular line editors. It doesn't use the &.

Minuses: The more commands there are, the more you need a reference card. It should have been included to make the package complete.

Documentation: A clearly written, helpful guide to making the most from GALE is included (53 pages) in an easy-to-read manner.

Skill level: Some programming expertise is desirable to make the best use of GALE.

Reviewer: Phil Daley

Product Name: **LOGO**
Equip. Req'd: Commodore 64 and 1541 Disk Drive
Price: $49.95
Manufacturer: Commodore Business Machines Inc.
1200 Wilson Drive
West Chester, PA 19380

Description: This is a fairly extensive implementation of LOGO (a procedural language developed at M.I.T.). Supplied on a single disk, it includes system primitives (commands) for graphics, arithmetic & logical operations and list processing. A second disk containing instructive demos, games and various utilities is also included. Most notable among the various utilities is a LOGO assembler which facilitates the addition of assembly language extensions to the language.

Pluses: This is a powerful language which is suited to many levels of application. At the bottom level it is almost ideally suited for entertaining and teaching children logical thought and expression. At a higher level it is a good vehicle for the study of structured recursive programming. At the top level, the list processing capabilities make LOGO a suitable candidate for implementing AI (Artificial Intelligence) concepts on a micro computer.

Minuses: LOGO is fairly large and complex (compared to BASIC). It was apparently necessary to cut a few corners in order to implement it on micro's. One indication of this is the fact that the garbage collection routines do not function properly. It's possible for lists of unused words or procedure names (usually resulting from typo's) to accumulate and reduce the available workspace. This defect will only be noticed during long program development sessions.

Documentation: In addition to the demo's and examples on the utilities disk, a 400+ page manual is provided. This manual contains major sections on graphics, computation, and list processing. It also covers sprites and sound/music generation. Extensive appendices cover assembly language programming and contain a complete glossary of LOGO primitives. The manual is good as a tutorial, but leaves something to be desired in conciseness and accessibility for quick reference purposes.

Skill level: The skill required depends on which level you approach LOGO. Little skill is required to "drive" the turtle around the graphics screen. More is required to write concise structured programs, and considerable skill is required to implement AI constructs.

Reviewer: Roger C. Crites

Product Name:	**Advanced X-tended Editor**
Equip. Req'd:	Apple II
Price:	*
Manufacturer:	Versa Computing, Inc. 3541 Old Conejo Rd. Newbury Park, CA 91320

Description: AXE is an Applesoft line editor which includes many time saving features for BASIC program development. Search and replace, auto line number, memory status, monitor commands, special formatted listings and line editing features are all available with AXE running. Editing commands include insert, delete, gobble, copy, uncopy, lower case, verbatim mode and a complete macro definition and table use for single key entry of often used strings.

Pluses: AXE appears transparent to the user and is a great help in editing lines without the POKE 33,33 routine. Search and replace strings are easily defined and are very useful in locating and changing variable names.

Minuses: A quick reference card is needed to help in remembering the different commands. I had some trouble in listing programs full width to the printer while AXE was active.

Documentation: The 50 page manual is well written and clearly explains how the various commands operate.

Skill level: Some experience with BASIC programming is necessary to derive all the benefits.

Reviewer: Phil Daley

Product Name:	**The Oddsmaker**
Equip. Req'd:	Commodore 64 or Apple II Disk Printer optional
Price:	$44.95
Manufacturer:	CZ Software 358 Forest Road South Yarmouth, MA 02664

Description: This program could be called "The Electronic Bookie", for that is exactly its function! Through an easy-to-use menu driven system you take bets on some activity, calculate and display the para-mutual odds, display the amount bet on each team/horse/fighter, print tickets for each bet, and when the contest is over, display the pay-offs for each bet. Additional features include automatically taking a 'house cut' percentage from each bet and saving the betting data to disk. The program is so complete that the creators hope they do not get in trouble with 'you-know-who'!

Pluses: Easy to use by anyone. Provides a good understanding of the para-mutual betting process - quite educational. The printed tickets feature makes the package really useful (for fun only, of course!).

Minuses: Perhaps overpriced at $44.95.

Documentation: The twenty page booklet is clearly written and easy to use.

Skill level: Can be used by anyone.

Reviewer: Robert M. Tripp

Product Name:	**BASIC Tutor**
Equip. Req'd:	Apple II+ or IIe, 1 Drive
Price:	*
Manufacturer:	Courseware Applications Savoy IL (c) 1983
Distributor:	SuperSoft P.O. Box 1628 Champaign, IL 61820

Description: A course in BASIC programming with lessons and exercises on disk and in the manual. Covers: what is programming, variables, expressions, entering & editing code, output (PRINT, TAB), input, branching (IF-THEN, GOTO), looping (FOR-NEXT-STEP), and READ-DATA. Material is fairly sophisticated. Can be used in a classroom or for self-study.

Pluses: Quality of presentation is quite good. Overall program is well designed. Covers all major BASIC commands. Examples not trivial, as in many BASIC teaching programs.

Minuses: Interaction is weak and inconsistent: sometimes when you answer incorrectly, you are shown the answer immediately; other times, you get 2 or more tries. You are not forced to enter the correct answer (to show that you accept it) before proceeding. Does not protect against keybounce - if you press RETURN several times quickly, you may flash through the next screen(s) of info without time to read.

Documentation: Superb manual with lesson outlines & goals, recaps of disk lessons, additional info, summaries, problems with solutions, reference list of disk commands, and glossary. Material presented well for older audiences; too many words per page for 12 yr olds to absorb (plus adult vocabulary).

Skill level: Some reasoning & problem-solving skills; age 15-adult.

Reviewer: Mary Gasiorowski

Product Name:	**Card?**
Equip. Req'd:	Commodore 64 or VIC 20 and a parallel (Centronics cable) printer
Price:	$75
Manufacturer:	Cardco, Inc. 313 Matthewson Wichita, KS 67214

Description: A printer interface to print text and graphics from your Commodore computer to any parallel printer. The included cables plug into the computer's cassette and disk drive ports without interfacing with those devices. Internal dip switches allow for permanent selection of features and software selection is also available. Card? features ASCII conversion, graphics printing (if your printer allows it), and a listing mode that converts color change/cursor move functions to understandable abbreviations. Several appendices will tutor you in screen dumps, printer control characters, and device selection.

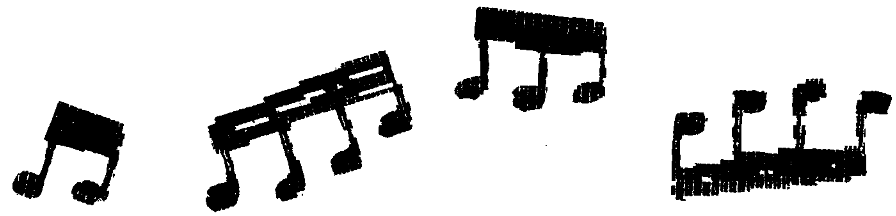
Pluses: Card?'s flexibility is its chief asset and the newest version supports Epson Graftrax +. Setup is easy and the instructions form a useful tutorial.

Minuses: Interfacing Card? with word processors can become complex if the program attempts ASCII conversion prior to sending the data through Card?. The problems arise when you attempt to imbed printer commands in the text. However, Cardco, Inc. provides suggestions and promises technical support to overcome these obstacles.

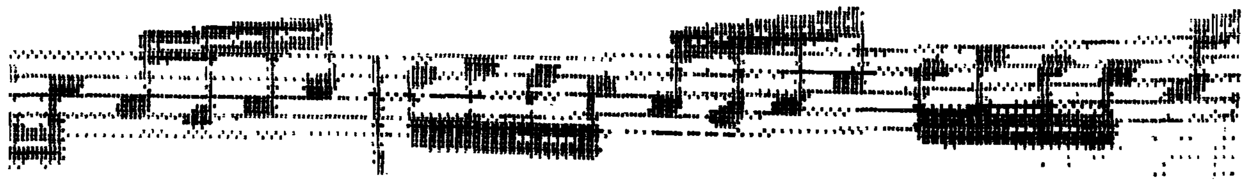
Documentation: A new booklet is in the works for the Graftrax update. Until then an addendum fills the gap. The instructions are detailed with intelligent examples and should answer your questions.

Skill level: Recommended for intermediate and advanced users only. If you don't know what ASCII conversion is you'll have trouble taking advantage of Card?'s features.

Reviewer: Mike Cherry



Musical Notes for the Apple



by Phillip Bowers

With a 200 note table and a five octive range covering the entire treble, bass and alto clefs, you can do more than just whistle 'Dixie'.

The greatest limitation I found with my APPLE II+ (APPLESOFT) is its inability to produce a variety of sounds. To overcome this I wrote a small assembler program (28 bytes long) that is CALLed by a Basic program using at least one POKE command.

Once loaded, the assembler program is capable of producing tones from 1.54hz to well over 15,000hz (cycles per second). Any tone can be held, from several seconds for very high tones, to minutes for lower tones. Four bytes are used in page zero (from $FC to $FF: 252 to 255) to control both the frequency of the tone and its length. The first two bytes set the frequency, and the last two limit the lingth (time) of the tone. Because two bytes are used for each, their combined values range from 1 to 65024 ($01 to $FE00). The use of the values 0($00) and 255 ($FF) are restricted by the relationship between the two programs.

By using page zero, it allows the assembler program to be located anywhere in RAM where it will be safe. For our purposes here the assembler program will be at address $6000, and the variable ASSEM, in the Basic program, will equal 24576. Once the values are set in page zero, the Basic program uses the command

CALL ASSEM

to produce the desired tone. Moving the assembler program only requires that the variable ASSEM be set to the decimal equivalent of the assembler's starting address.

The assembler program can reproduce any musical note, including sharps/flats, up to G# below A of 880hz. Above 880hz the rounding errors for many notes are too great to be of much use. Up to 880hz the largest error is 2.74 cycles, at G and G# just below 880hz. As the frequency decreases, so does the error in cycles.

The useable musical notes are from G# below 880hz, to A at 27.5hz. This gives a 5 octive range covering the entire treble, bass, and alto clefs. The Basic program will allow a 10 octive range of inputs, from 0 to 9, but octives 1 to 5 cover the above clefs, with 1 being the lower octive.

To setup the assembler program enter Monitor

CALL · 151

then the RETURN key. Once you have the asterisk prompt, enter the following after it:

**6000:A5 FE 38 EA A6 FC AE 30
CO A6 FC A4 FD CA DO FD
88 DO FA E9 01 DO EB C6
FF DO EB 60 00**

then RETURN.

The above should be entered as one continuous string, with each byte seperated by a space. It is shown as it would basically appear in a memory dump. A memory dump is done by entering

6000.601C

By entering

6000L

you will get an assembler listing. The listing is reproduced as a debugging aid.

```
6000- A5 FE       LDA $FE
6002- 38          SEC
6003- EA          NOP
6004- A6 FC       LDX $FC
6006- AE 30 CO    LDX $CO30
6009- A6 FC       LDX $FC
600B- A4 FD       LDY $FD
600D- CA          DEX
600E- DO FD       BNE $600D
6010- 88          DEY
6011- DO FA       BNE $600D
6013- E9 01       SBC #
6015- DO EB       BNE $6002
6017- C6 FF       DEC $FF
6019- DO E8       BNE $6006
601B- 60          RTS
601C- 00          BRK
```

To save this to disk, use the following:

BSAVE ASSEM SOUND, A$6000, L$1C

The length shown ($1C) will not save the last byte at $601C; it is included to show the ending address only.

Still in Monitor, enter

FC:F9 02 07 18

then
6000G

when the RETURN key is pressed it will execute the assembler program using the values at $FC to $FF. The note should be G below middle C, and last 15 seconds. After the assembler program has executed, the value at address $FF should be equal to 0 ($00), the other values should be untouched. If your values differ, check for an error.

For A below middle C:
FC:C2 02 D9 1A

for D above middle C:
FC:50 02 81 23

these values should all give 15 second notes when the assembler program is executed.

After the "Musical Notes" program is entered the keyboard keys "QWERTYU"should be marked to read "ABCDEFG", respectively. The sharp/flat of a note is obtained by pressing the "CNTL" key and the note key together. A rest note is given by the space bar.

As each note is entered, it will be stored in an internal table for replaying, and displayed to the upper 20 text lines. The note table (NT) holds the values for the note frequency and its length in a low order, high order format that are POKEed directly into addresses $FC to $FF. Each note displayed to a text line will have the format "ABCb", where:

A = the octive the note is in (0 to 9).
B = the note value, A through G.
C = the note time;
W : whole
H : half
4 : fourth
8 : eighth
1 : sixteenth
3 : thirty-second
6 : sixty-fourth
b = space, note separator.

A sharp/flat of the note will have the same format, except that it will be in the INVERSE mode. A rest note will be displayed as "bRCb". The "R" meaning a rest note, and the "C" the rest time.

This format allows 10 notes per text line, and by using the first 20 text lines, it permits 200 notes to be displayed. The last 4 text lines are for program information.

When the program is executed, the first input line will be:

ENTER BEATS PER MINUTES (4TH NOTE)

If no value is entered, the beats per minute will be set to 120. Some sheet music will have a note symbol (like a quarter note), followed by an equal sign, then a number in parenthesis near the upper left corner. This number is the beats per minutes. Whatever value is entered at this point will be the base for timing all other notes.

Once the beat is entered, you are ready to start entering notes. To change the octive, press any of the numeric keys (0 to 9). Line 22 (VTAB 22) will show the current octive.

The "BEAT =" shows the base beat of 120 at a quarter note value. To change the beat for any note, or group of notes, the keys "ASDFGHJ" are used.

A = whole note.
S = half note.
D = fourth note.
F = eighth note.
G = sixteenth note.
H = thirty-second note.
J = sixty-fourth note.

By pressing the "A" key, the beat will change from:

BEAT = 120 AT 4TH = 120

to:

BEAT = 120 AT WHOLE = 30

the last number is the beats per minute for a whole note.

Once a note is entered it cannot be deleted, but it can be changed to any value. The ", " and "." keys are used to move the cursor over any note you want to change. The "," will move the cursor to the left, lower in the note table, while the "." will move it to the right, but will not allow movement beyond the next enterable note position.

While the left and right arrow keys would have been better, they were not used because the right key has the same value as the "CNTL" and "U" keys, which would give the note G#.

The option "Z : RUN NOTES (0)" shows how many notes are currently being stored, and will run all notes regardless of the cursor position.

In the event either the "X" or "C" is pressed, you will be asked if you really want it before they do their thing. Option "X" will CLEAR everything, and restart the program to

Listing 1

```
5   GOSUB 255: HIMEM: ASSEM
10  VTAB 21: PRINT "ENTER NOTE OR OPTION :                    ":
    PRINT "OCTIVE="OL" BEAT="BM" AT "BV$" = "BM / BV"  "
15  PRINT "Z:RUN NOTES("NO")" TAB( 20)"X:NEW NOTES": PRINT "C:
    END PROGRAM"
20  VTAB VT: HTAB HT: GET IN$:ER = 0:KI =  ASC (IN$): GOSUB 100:
    IF ER = 0 THEN VTAB 23: HTAB 1: GOTO 15
25  IF KI = 65 THEN BV$ = "WHOLE":BV = 4: GOTO 70
26  IF KI = 83 THEN BV$ = "HALF":BV = 2: GOTO 70
27  IF KI = 68 THEN BV$ = "4TH":BV = 1: GOTO 70
28  IF KI = 70 THEN BV$ = "8TH":BV = .5: GOTO 70
29  IF KI = 71 THEN BV$ = "16TH":BV = .25: GOTO 70
30  IF KI = 72 THEN BV$ = "32ND":BV = .125: GOTO 70
31  IF KI = 74 THEN BV$ = "64TH":BV = .0625: GOTO 70
35  IF KI >  = 48 AND KI <  = 57 THEN OL = KI - 48:
    HTAB 1: GOTO 10
40  IF KI = 44 AND NP - 1 >  - 1 THEN NP = NP - 1:
    GOSUB 225: GOTO 20
```

Listing 1 *(continued)*

```
45  IF KI = 46 AND (NP < NO) THEN NP = NP + 1: GOSUB 215: GOTO 20
50  IF KI = 90 THEN  GOSUB 200
55  IF KI = 88 OR KI = 67 THEN 180
60  GOTO 20
70  SB = (60 / BM) * BV:TM =  INT (((SB * (1E + 6)) / 36372) + 0.5):
    HTAB 1: GOTO 10
100  SV = OL: IF KI = 32 THEN OL = 0:XF = 0: GOSUB 145:
     IN$ = " R" +  LEFT$ (BV$,1) + " ": PRINT IN$:
     NT(NP,0) = FL * ( - 1):NT(NP,1) = FH:NT(NP,2) = LL:
     NT(NP,3) = LH:OL = SV: GOSUB 215: GOSUB 170: RETURN
105  FOR X = 0 TO 11: IF KI = KV(X,0) THEN NV = KV(X,1):
     XF = X:X = 98
110  NEXT : IF X = 12 THEN ER = 1: RETURN
115  GOSUB 145:NT(NP,0) = FL:NT(NP,1) = FH:NT(NP,2) = LL:
     NT(NP,3) = LH
120  IN$ = "":IN$ =  STR$ (OL) +  MID$ (KL$,NV,1) +
     LEFT$ (BV$,1) + " ": IF KI < 32 THEN  INVERSE
125  PRINT IN$: NORMAL : GOSUB 215
130  POKE 252,FL: POKE 253,FH: POKE 254,LL: POKE 255,LH:
     CALL ASSEM
135  IF (NP + 1 > NO) THEN NO = NO + 1:NP = NP + 1: RETURN
140  NP = NP + 1: RETURN
145  OC = (2 ↑ OL) * (2 ↑ (XF / 12)):CY = SC * OC:TC = TM * OC:
     PS = 1E + 6 / (2 * CY)
150  FH =  INT ((PS + 1254) / 1279):FT = 21 + (5 * FH - 1) +
     (1274 * (FH - 1)):FL = INT (((PS - FT) / 5) + .5)
155  TI = TC:LH =  INT (TI / 255):LL = (((TI / 255) - LH) * 255):
     IF LL = 0 THEN LL = 1
160  LH = LH + 1
165  RETURN
170  IF (NP + 1 > NO) THEN NO = NO + 1:NP = NP + 1: RETURN
175  NP = NP + 1: RETURN
180  IN$ = "END": IF KI = 88 THEN IN$ = "NEW"
185  VTAB 21: HTAB 1: INVERSE : PRINT "ENTER 'Y' FOR "IN$",
     ANY KEY TO IGNORE. ";: GET IN$: NORMAL :
     IF IN$ <  > "Y" THEN  HTAB 1: GOTO 10
190  IF KI = 67 THEN 300
195  CLEAR : HOME : GOSUB 260: GOTO 10
200  IF NO = 0 THEN  RETURN
205  FOR X = 0 TO NO - 1: POKE 252, ABS (NT(X,0)):
     POKE 253,NT(X,1): POKE 254,NT (X,2): POKE 255,NT(X,3):
     IF NT(X,0) < 0 THEN  POKE ASSEM + 8,0
210  CALL ASSEM: POKE ASSEM + 8,192: NEXT : RETURN
215  HT = HT + XI: IF HT = 41 THEN HT = 1:VT = VT + 1:
     IF VT = 21 THEN  GOSUB 235
220  RETURN
225  HT = HT - XI: IF HT < 1 THEN HT = 37:VT = VT - 1:
     IF VT = 0 THEN VT = 1:HT = 1
230  RETURN
235  INVERSE : VTAB 21: PRINT "TABLE FULL !! ANY KEY
     TO CONTINUE. ";: GET IN$: NORMAL
240  NP = NP - 1:VT = VT - 1:HT = 37: IF NO < 200 THEN NO = NO + 1
245  RETURN
255  HOME : PRINT "MUSICAL NOTES FOR THE APPLE": PRINT :
     PRINT "BY PHILLIP BOWERS": PRINT :
     PRINT "ROCHESTER, N.Y.":PRINT
260  BM = 120: INPUT "ENTER BEATS PER MINUTE (4TH NOTE) ";B$:
     SB =  VAL (B$): IF SB > 0 THEN BM = SB
265  BV$ = "4TH":BV = 1:SB = 60 / BM:
     TM =  INT (((SB * (1E + 6)) / 36372) + 0.5)
270  ASSEM = 24576: DIM KV(11,1): DIM NT(199,3): FOR X = 0 TO 11:
     READ NO,NP:KV(X,0) = NO:KV(X,1) = NP: NEXT :KL$ = "ABCDEFG"
275  VT = 1:HT = 1:XI = 4:NO = 0:NP = 0:ST = 13.75:SC = ST:OL = 1
280  X =  PEEK (ASSEM): IF X <  > 165 THEN IN$ =  CHR$ (4):
     PRINT IN$;"BLOAD ASSEM SOUND"
285  HOME : RETURN
290  DATA  81,1,17,1,87,2,69,3,5,3,82,4
295  DATA  18,4,84,5,89,6,25,6,85,7,21,7
300  VTAB 21: HTAB 1:
     PRINT "PROGRAM END ROUTINE                    ": END
```

the initial beats per minute.

Option "C" does not directly END the program, rather it passes control to line 300. The lines 300 through end have been left open so that you can save the note table, or whatever else you may want to do before ENDing.

Any key other than those mentioned above will be ignored. The program will just continue along its merry way.

The note table is defined in line 270

DIM NT(199,3)

where

NT(NO,0) = low order value for the NT(NO,1) = high order value for the note.
NT(NO,2) = low order value for the note length.
NT(NO,3) = high order value for the note length.

The current number of table entries is equal to NO - 1. The value of ASSEM is also set in line 270.

Even though the number of entries can be made greater than 200, it is not suggested because you will lose the relationship between the screen notes and the notes in the note table once more than 200 notes are entered.

In conclusion, I would like to point out that the note table (NT) uses over 9 times more space than is necessary to store the note values and their lengths. This is because we are using an array defined by a Basic program. Because of this it is not possible to use HGR (page 1). While the basic program uses about 2400 bytes, the note table requires an additional 9600 bytes to save the 800 bytes needed by the assembler program.

While it is possible to POKE these values directly into RAM, it should be noted that it will actually require 1000 bytes to store the data. An additional byte is needed for each note to indicate a rest value. In the basic program a rest note uses the same 4 data bytes as any other note, except that the low order rest value is negative (NT(NP,0))...line 100 in the program.

When the notes are replayed the absolute value (ABS) value is POKEed into address $FC (252), and the assembler program is altered so as not to reference the speaker location when a negative value is encountered. But it is executed in the same manner as any note. So if you decide you need more space, or want to use HGR, then remember to include the additional byte for each note.

MICRO

Under the 64 ROM

by John A. Winnie

Requirements: Commodore 64

Free up your BASIC memory without sacrificing program messages using the 16K of RAM under the BASIC and Kernal ROM chips.

Although the Commodore 64 has a hefty chunk of free BASIC memory (38911 bytes at power-up), sometimes it can still turn out that additional memory will make the difference between a polished program and dull code seriously weakened by compromises. In many programs the chief memory-muncher is the string data: the various descriptions and messages that eat up BASIC bytes by just being in the program, and then go on to cost even more when they are accessed by arrays. A good adventure game, for example, may inflict hundreds of different messages on its player ("You can't take that. It's tied down."), and if these are all stored in the BASIC programming area, valuable programming space is lost.

The program presented here, called "Printout", solves the problem of string data storage simply and economically. Although it is written in machine language , it is unnecessary to know machine language in order to use it most effectively; however, it is a good idea to know just what it does.

What Printout Does

Between them, two ROM (read-only memory) chips in the 64 use up to 16K of what would otherwise be free RAM. The first chip contains the 64's version of BASIC and lies over memory addresses 40960 through 49151 ($A000-$BFFF). The second ROM chip contains the operating system of the 64 and is called the "Kernal". It covers memory addresses 57344 to 65535 ($E000-$FFFF). Since the first chip contains BASIC and the Kernal ROM contains the operating system's machine code routines, it seems that the 16K of RAM has been sacrificed to some good purpose--and, of course, this is quite true. Remarkably, however, much of the sacrifice can--with a little finagling--be avoided altogether.

First of all, data may be placed in these locations in the usual ways: by direct pokes from BASIC, for example, or by loading a file straight into the under-ROM area. The trick is to get the data out once it has been stuffed in. A PEEK to any of these locations, for example, will read the contents of the ROM chip at that address, not what is stored in the underlying RAM location.

Fortunately, there is a way around the problem. Both ROM chips may be switched out by a simple poke (POKE1,52), exposing the underlying RAM in all its glory! Peeking is now added to poking--or would be, except for one thing: with BASIC so cavalierly switched away, so too for PEEKing! This is why we need machine language to finally solve the problem. We can switch off the two ROM chips using BASIC, but we need machine language to access the now-exposed RAM, and, when we are through with that, switch us back again to BASIC.

Now Printout does all this and more. Once the ROM chips have been switched out, Printout prints to the screen any messages that have been stored under the ROM chips. Of course, the messages must be stored there in the appropriate form. First of all, each message must be surrounded by zeroes; the message itself is coded by simply using the ASCII number of each of its characters. Thus the sequence:

```
  H  E  L  P  !     E  R  R  O  R
Ø,72,69,76,8Ø,33,Ø,69,82,82,79,82,Ø,
```

when stored in memory locations 40960 through 40972, encodes the two messages: "HELP" and "ERROR". When strings are stored in this way, all that Printout needs to know is which message you would like printed out (counting 0,1,2,...), and where your block of messages begins (in this example, at 40960). So to use Printout, POKE the message number into memory location 2 (decimal), and the low and high bytes of the base address of the message block into locations 251 and 252 (decimal), respectively. (Much of this is done for you by subroutine 50000 in the program Printoutloader of Listing 1).

Using PRINTOUT

Listing 1 is a BASIC loader for Printout. After adding it to your program, a call to subroutine 60000 loads Printout into memory locations 828 through 883. The other subroutine included (50000) may now be called when a message is to be printed. It needs to be supplied with only two pieces of information. First, the base address of the block where your messages are stored; this is the value of the variable ADD. And second, the message number must be supplied; this is the value of the variable ME. When subroutine 50000 is now called, the ME-th message will be printed, beginning at the current cursor position. (Normally, that cursor position will be set by the rest of the program before calling this subroutine). The load address (828 decimal) in lines 50010 and 60002 of Listing 1 (and Listing 3) is not critical. Since the machine code is relocatable, any free area of RAM may be used to hold Printout's 56 bytes.

Of course, in order to use Printout, messages must be previously stored under the BASIC or the Kernal ROMs. An easy way to do this is to create a program file of these messages, and then load this file at the beginning of your program. Listing 2, Messagewriter, is designed to create such files. In line 20 you specify the total number of messages (minus one), and in line 25 you specify the base address of this block of messages. You supply the actual messages in the data statements beginning at line 500. Since you will need to keep track of your messages and their numbers, Messagewriter also generates a numbered, hardcopy list of your messages.

Listing 3 provides an example of using Printout to list the messages used earlier in Listing 2. It assumes that you already have run the program of Listing 2 and have the program file "Messages" on your disk. Although the program of Listing 3 does not do anything spectacular, it does wrap up all that has come before. If you understand how it works, then the power of Printout and the new 16K that comes with it is at your fingertips! One more thing. Since Printout places no restrictions on string length, an entire screen may be stored under ROM as a single 999 byte string. When Printout is called, the stored screen is displayed almost instantly, certainly much more rapidly than when a screen is loaded from a disk file.

To the Machine Language Beginner

As the assembly listing shows, Printout is in general quite straightforward. The one slightly tricky thing is that it uses a Kernal ROM routine (CHROUT, at $FFD2) on data stored under the Kernal ROM itself. So the Kernal, after first being switched off to permit access to the character data, is next switched back on to permit the Kernal routine CHROUT to print out the character. Next--and here is the tricky part--the Kernal is switched back off again to get the next character. But CHROUT, it happens, restores the hardware interrupts along its way! Should such an interrupt now take place while the Kernal ROM is switched out, the system will crash, since the interrupt routines are themselves Kernal ROM routines. Hence the added step (SEI) to repeatedly disable the hardware interrupts each time CHROUT is used. The moral should be clear: even though interrupts have been disabled initially, each time a Kernal routine is used--any Kernal routine--the safest bet is to again disable interrupts before going on to switch off the Kernal ROM.

PRINTOUTLOADER

```
50000 REM * PRINTOUT SUB *
50005 REM * INPUTS ARE ME AND ADD *
50010 POKE 2,ME:HB=INT(ADD/256):LB=ADD-256*HB:
      POKE 251,LB:POKE 252,HB:SYS 828:RETURN
60000 REM * LOAD PRINTOUT DATA SUB *
60002 FOR I=828 TO 883:READ Q:POKE I,Q:NEXT:RETURN
60005 DATA 120,169,52,133,1,162,255
60010 DATA 160,255,198,252,232,200,208
60015 DATA 2,230,252,177,251,240,2
60020 DATA 208,245,228,2,208,240,200
60025 DATA 208,2,230,252,177,251,208
60030 DATA 6,169,55,133,1,88,96
60035 DATA 162,55,134,1,32,210,255
60040 DATA 120,169,52,133,1,208,227
```

MESSAGEWRITER

```
10 REM * MESSAGEWRITER *
20 NMESS=5:REM * NUMBER OF MESSAGES -1 *
25 ADD=57344:REM * BASE OF MESSAGE BLOCK *
30 RESTORE:PRINT"{CLEAR,DOWN10}"
   TAB(10)"PRINTOUT OR FILE(P/F)?"
35 GET A$:IF A$=""OR(A$<>"P"ANDA$<>"F") GOTO35
37 PRINT"{CLEAR,DOWN10}"TAB(15)"THANK YOU."
40 IF A$="P"THEN GOSUB 100:GOTO 30
45 REM * WRITE MESSAGE FILE *
50 OPEN 15,8,15:PRINT#15,"S0:MESSAGES"
60 OPEN 5,8,5,"0:MESSAGES,P,W"
65 HB=INT(ADD/256):LB=ADD-256*HB
70 PRINT#5,CHR$(LB)CHR$(HB);:
   REM * FILE WILL LOAD AT ADDRESS = ADD *
75 FOR I=0 TO NMESS:READ D$:L=LEN(D$)
80 PRINT#5,CHR$(0);
85 FOR J=1 TO L:PRINT#5,MID$(D$,J,1);
90 NEXT:NEXT
95 PRINT#5,CHR$(0);:CLOSE 5:CLOSE 15:GOTO 30
100 REM * PRINTOUT SUB *
110 OPEN 1,4:PRINT#1,CHR$(14)"MESSAGE LIST"CHR$(15)
120 FOR I=0 TO NMESS:READ D$:PRINT#1,I,D$:NEXT
130 PRINT#1:CLOSE1:RETURN
500 DATA HELLO THERE,YOU ARE IN A DARK CAVERN
510 DATA WHY NOT?,THAT WAS VERY FOOLISH
515 DATA STOP RIGHT THERE!,
    YOU HAVE BEEN KILLED. TRY AGAIN?
```

MESSAGE DEMO

```
5 REM * MESSAGE DEMO *
10 IF L=0 THEN L=1:LOAD "MESSAGES",8,1
20 GOSUB 60000:REM * LOAD PRINTOUT *
100 ADD=57344:REM * BASE ADDRESS OF MESSAGE BLOCK *
105 NMESS=5:REM * (NUMBER OF MESSAGES)-1
110 FOR I=0 TO NMESS:ME=I:GOSUB 50000:PRINT CHR$(13):NEXT
120 GOTO 110
50000 REM * PRINTOUT SUB *
50005 REM * INPUTS ARE ME AND ADD *
50010 POKE 2,ME:HB=INT(ADD/256):LB=ADD-256*HB:
      POKE 251,LB:POKE 252,HB:SYS 828:RETURN
60000 REM * LOAD PRINTOUT DATA *
60002 FOR I=828 TO 883:READ Q:POKE I,Q:NEXT:RETURN
60005 DATA 120,169,52,133,1,162,255
60010 DATA 160,255,198,252,232,200,208
60015 DATA 2,230,252,177,251,240,2
60020 DATA 208,245,228,2,208,240,200
60025 DATA 208,2,230,252,177,251,208
60030 DATA 6,169,55,133,1,88,96
60035 DATA 162,55,134,1,32,210,255
60040 DATA 120,169,52,133,1,208,227
```

Listing 1

```
                        ; PRINTOUT STRINGS STORED UNDER BASIC
                        ; OR UNDER THE KERNAL.
                        ;
                        ; STRINGS ARE STORED IN THE FORM:
                        ;  Ø,-,-,-,Ø,-,-,-,-Ø,—   ETC.
                        ;
                        ; THE STRING PRINTED IS THE N-TH STRING STORED
                        ; IN THAT BLOCK OF STRINGS WHICH BEGINS AT THE
                        ; BASE ADDRESS POINTED TO BY BASELO,BASEHI.
                        ;
                        ; THE STRING NUMBER N IS PREVIOUSLY POKED
                        ; INTO MEMORY LOCATION 2 (='STRING').
                        ;
Ø828                    ;        ORG $828     ; RELOCATABLE
                        ;
ØØØ2                    STRING   EQU 2
ØØFB                    BASELO   EQU 251
ØØFC                    BASEHI   EQU 252
                        ;
Ø828 78                          SEI           ; DISABLE INTERRUPTS
Ø829 A9 34                       LDA #52       ; OUT KERNAL/BASIC
Ø82B 85 Ø1                       STA $Ø1
                        ;
Ø82D A2 FF                       LDX #255      ; SET CNTRS
Ø82F AØ FF                       LDY #255
Ø831 C6 FC                       DEC BASEHI
                        ;
Ø833 E8                 COUNT    INX
Ø834 C8                 HUNT     INY
Ø835 DØ Ø2                       BNE GTBYTE    ; NOT PAGE END? GO ON.
Ø837 E6 FC                       INC BASEHI    ; PAGE END. NEXT PAGE.
                        ;
Ø839 B1 FB              GTBYTE   LDA (BASELO),Y      ; GET BYTE
Ø83B FØ Ø2                       BEQ CHECK     ; A ZERO. THE RIGHT ONE?
Ø83D DØ F5                       BNE HUNT      ; NOT ZERO. KEEP HUNTING.
                        ;
Ø83F E4 Ø2              CHECK    CPX STRING    ; THE RIGHT ZERO?
Ø841 DØ FØ                       BNE COUNT     ; NO? FIND NEXT ZERO
Ø843 C8                 PRINT    INY           ; READY TO PRINT
Ø844 DØ Ø2                       BNE OUTPUT    ; NOT END OF PAGE? PUT IT OUT.
Ø846 E6 FC                       INC BASEHI    ; END? TURN PAGE.
Ø848 B1 FB              OUTPUT   LDA (BASELO),Y      ; GET CHARACTER
Ø84A DØ Ø6                       BNE CHROUT    ; NOT END OF BLOCK? PRINT IT.
Ø84C A9 37                       LDA #55       ; END OF BLOCK. RESTORE KERNAL.
Ø84E 85 Ø1                       STA $Ø1
Ø85Ø 58                          CLI           ; RESTORE INTERRUPTS
Ø851 6Ø                          RTS           ; DONE!
                        ;
Ø852 A2 37              CHROUT   LDX #55       ; RESTORE KERNAL/BASIC
Ø854 86 Ø1                       STX $Ø1
Ø856 2Ø D2 FF                    JSR $FFD2     ; CHROUT(KERNAL)
                        ; TRICKY! $FFD2 RESTORES INTERRUPTS,
                        ; SO THEY MUST BE DISABLED AGAIN.
Ø859 78                          SEI
Ø85A A9 34                       LDA #52       ; TAKE OUT KERNAL
Ø85C 85 Ø1                       STA $Ø1
Ø85E DØ E3                       BNE PRINT     ; NEXT CHARACTER
                        ;
Ø86Ø                             END
```

feature

A Better Random Number Generator

by H. Cem Kaner and John R. Vokey

Reap the fruit of 7 years of labor—a superior version of the random number generator, for simulations, gambling, forecasting.

NOTE: The work for this article was supported by a research grant from the Natural Sciences and Engineering Research Council of Canada (NSERC) to Dr. A.B. Kristofferson, and by NSERC Postgraduate Scholarships to the authors. The authors would also like to thank John Lyons for the many helpful discussions of RNG design.

Random number generators (RNGs for short) are functions that produce pseudo-random numbers. Usually the numbers produced are fractions between 0 and 1. Ideally, a computer language's RNG should be able to generate every fraction that the language can represent, every fraction should be as likely to be generated as every other, and the order of the numbers should be completely unpredictable to the user. Slightly more formally, the RNG should produce sequences of numbers which, so far as standard statistical tests can tell, behave in the same way as "truly random" number sequences would behave.

RNG's are used to simulate imperfectly predictable real life events. Computer games use them in this way. So do some insurance companies, when setting rates. Economists, psychologists, sociologists, consumer behavior researchers of various backgrounds, often work with theories of such complexity that the only way that they can decide whether a theory is correct is to simulate the behavior of a population on the computer, and to compare this with the actual behavior shown by the groups they are studying. Gamblers use random number generators to "shuffle" cards or "roll" dice. They try different betting strategies at the computer, where it's free, rather than at the casino (or the stock market), where they can lose their shirt. Simulation involving random number generators is often called Monte Carlo simulation, after the casinos in Monte Carlo: much of the early research on probability and statistics was financed by gamblers. As final example of simulations, estimates of the likelihood of an accident in a nuclear reactor, and of its probable severity, are often made by simulation, before the reactor is built, to check if safety measures are adequate.

RNG's are also used to provide random test data for input to complex computer programs. It is impossible to test every branch or path in a major program. Random inputs or combinations of inputs often expose bugs that a systematic selection of test cases missed.

In this article we present an assembly language program, interfaced to Applesoft via the USR function, which provides three independently addressable RNGs. Because there is so little available in relatively nontechnical language about RNGs, and because of their growing importance, we will also describe how we chose them. Finally, we will outline some of the tests that we performed on them. The quality of a random number generator is not determined by the elegance of its code, but instead by the randomness of the sequences of numbers that it produces. The test results are always the most important part of the documentation of any RNG.

The RNG Algorithm

There are many ways to produce pseudo-random numbers, a few of which work reasonably well. Donald Knuth's excellent 178-page chapter on RNGs describes quite a few of all varieties. We use what is called a mixed linear congruential generator. Suppose that you store the numbers you generate in an array, so R[1] is the first number, R[2] is the second, and so on through R[N]. Let a, c and m be constants. We'll be concerned with their values later. The mixed linear congruential generator is defined by the following equation:

```
R[N+1] = (aR[N] + c) modulo m
```

In other words, if your last random number was R[N], your next one is obtained by multiplying R[N] by a, adding c to the product, and finding the value of that result modulo m. As usual (see the integer BASIC manual on MOD for more details) to obtain a number modulo m you divide it by m but keep the remainder rather than the quotient. For example:

```
13 mod 1Ø = 23 mod 1Ø = 972863 mod 1Ø = 3
```

A mixed congruential generator does not produce "truly random" numbers (no software RNG can) because it is possible, given knowledge of a, m and c to predict the next number from the last. However, if a and c are properly chosen and m is reasonably large, a person who did not know the formula, or even one who did know it but who didn't have a very good calculator handy, would be hard-pressed to predict the next number.

Selection of the RNG's Parameters

Not every mixed linear congruential random number generator is good. Most are terrible. The values of a, c and m determine how good the RNG will be. These three numbers are called the parameters of the generator. Different considerations are involved in choosing each number. Generally, m is chosen first, then a and c.

It is easy to find values of a and c which will guarantee that the RNG will

Numerical analysts work with RNGs to obtain numerical estimates of the solutions of complex mathematical functions for which no theoretical solutions exist, or to provide estimates against which a theoretical solution (which may be wrong) can be checked.

Randomization of the order of events in experiments, so that people (rats, whatever) cannot predict exactly what will happen next, has been a necessary part of the design of every experiment that we have run.

These are only some of the uses of random number generators: among other common ones are random sampling (for surveys and for quality control, for example), and partially random decision making (sometimes the best way to make an important decision, as studied in Game Theory).

The better the random number generator, the more lifelike or interesting the simulation, the stronger the test of the theory, the more likely the numerical solution is to be right, the more hidden bugs can be expected to be found in the program, the more valid the statistical test, the tighter the experimental control, the more representative the survey, the more unpredictable the decision.

Most implementations of high level computer languages provide something in their function library that the manual calls an RNG. Applesoft's RND function is typical of those we've seen on small systems. The reference manual describes RND as a source of random numbers, but it provides no evidence whatever that this claim should be believed, nor any warning that it should be taken with a mountain of salt.

RND, when subjected to standard statistical tests, fails them badly.

We should stress here that we are not singling out Apple for criticism. In our experience with various mini and microcomputers, manuals which admit to low-grade RNG's are nearly as rare as language implementations that provide an RNG worthy of the name.

It is not surprising that many languages' RNGs are poor. Much of the best research is very recent, conducted after some of these generators were written. Simulations require a great deal of computer time. They were not of general interest until computer time became very affordable.

produce every number between 0 and m-1 before the sequence starts to repeat itself. Eventually, no matter what a, c and m are, the series must repeat. How long it goes without repetition is called the period and the longest period that you can get with a congruential generator is m. For many reasons, the longer the period, the better the generator.

The second factor involved in choosing m involves computational convenience. As defined above, our RNG produces integers. To obtain fractions between 0 and 1, just divide these integers by m. Applesoft reserves 32 bits for the digits of any number. If we used m = 2^{32}, our sequence from 0 to m-1 would include every bit pattern than can be stored for a number. In general, since we are dealing with a binary computer, so numbers are stored as bit patterns, m should be a power of 2.

Unfortunately, m = 2^{32} will not yield every fraction that the Apple can store, because Applesoft uses an extra byte per number to hold the exponent. This allows representation of billions of different very small numbers, including numbers near 10^{-38}. Working with fractions of the form R[N]2^{32}, we can produce only one number in this region, namely zero. Tiny fractions in floating point languages are always under-represented by congruential generators: many fractions that the language can work with cannot be generated. We can alleviate the problem somewhat, and increase the period, by increasing m to 2^{40}. Not every possible fraction will be generated with this m -- R[N] would have to be 17 bytes long for that and the RNG would be very slow -- but when m is 2^{40}, R[N] can take on 1,099,511,627,776 different values, which is plenty. This is the value we use.

Our next decisions involve a and c and these are more difficult. It is easy to find values of a and c that allow the period to be m. If a mod 4 = 1 and c is any odd number, the period will be 2^{40} (i.e. m). But this is only part of the story.

As an absolutely awful example of a full period mixed linear congruential generator, suppose that a is 1 and c is also 1. So our generator is defined by

```
R[N+1] = (R[N] +1) modulo m
```

It works in the sense that we will indeed get all the numbers between 0 and m-1, but the sequence is 0, 1, 2, 3, etc., and this is hardly random.

The apparent randomness of a sequence of numbers is determined by the ordering of the numbers. This is where most RNGs, including all linear congruential generators, have at least some problems.

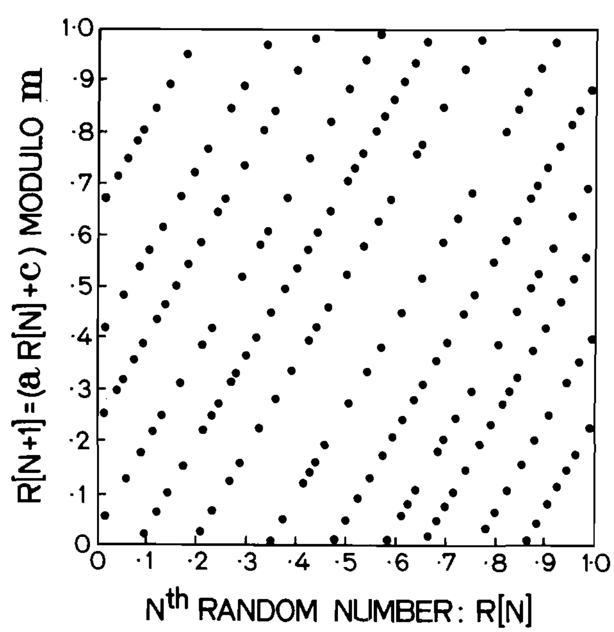


Figure 1

Linear patterning of successive pairs of numbers obtained from linear congruential random number generators. A good generator spreads the points across more lines, yielding as few as possible on each line. Nonlinear software generators exhibit nonlinear patterns in graphs of this type but the patterns are just as pronounced.

We can think about the ordering problem by thinking about short subsequences of the form (R[I], R[I1], R[I2], ..., R[IK]). Consider pairs first. There are m^2 possible pairs of numbers, (R[I], R[I1]), between 0 and m-1 but a generator of period m can only yield m different pairs. Which m pairs is the critical question.

In the case of R[I1] R[I] 1, a graph of R[I] along one axis and R[I1] along the other would show a single straight line.

A truly random sample of m numbers from the possible m^2 would result in points scattered all over the graph.

All linear congruential generators will produce graphs which show patterning, and that patterning will always be a set of parallel lines (see Figure 1). The trick is to find a generator which produces as many of these lines as possible, with as few points on each line as possible. The result will be a more even coverage of the m^2 possible pairs.

Note, by the way, that the larger m is, the more lines we can have and the closer they will be. The shorter the period, the poorer the generator.

A two dimensional graph, with R[I] on one axis and R[I1] on the other, is graph of a plane. A one-dimensional graph is simply a line. A three-dimensional graph, of a cube, contains planes just as a two-dimensional graph contains lines. If we plot sequences of three pseudo-random numbers, (R[I], R[I1], R[I2]), on a cube, all of the points will fall on parallel plane and all of the points on each plane will be on parallel lines. In this case, only m of a possible m^3 triplets can be produced by the RNG, so coverage of the cube is even more sparse than coverage of the plane in the two dimensional graph. The problem of patterning of triples is potentially more severe than patterning of pairs. In higher dimensions (longer sequences), we find parallel hyperplanes, and sparser and sparser coverage of the space.

We call this problem of patterning of linear congruential generators the lines and planes problem. Our goal is to minimize it. The more lines, planes, and hyperplanes we can cause our RNG to generate, the fewer the points on any given line, plane, etc., and the less patterning there is. In a truly random sequence, there is no patterning, and this is what we want to approximate with our pseudo-random sequence.

(If you are intrigued by this discussion but a little lost, George Marsaglia's chapter in the Encyclopedia of Computer Science is excellent and quite readable. Knuth's discussion of random numbers also deals with this problem at great length, with numerical examples and exercises. It is more technical, but in our opinion it is the best source available. For references to the original research, see Knuth).

The value of the RNG multiplier, a, is the main determinant of the degree of serial patterning. We want to choose a so as to produce as many lines and planes as possible, and to space them out as evenly as possible. This can be translated into the goal of minimizing the maximum distance between any two lines (planes, hyperplanes, etc.). Let 1/V2 be the maximum distance between any adjacent lines in a graph (such as Figure 1) of R[I1] against R[I]. Let 1/V3 be the largest distance between pairs of parallel planes in the graph of triplets (R[I], R[I1], R[I-2]), and so on. Our goal is to maximize V2, V3, V4, V5 and V6. (Note that these V's are inverses. The bigger the V, the smaller the largest distance between lines or planes in the graphs). We stop at 6 because if these values are good, higher dimensional sequential interactions are almost certainly unimportant. According to Knuth, we would be pretty safe stopping at 4.

The V values for linear congruential generators can be determined using a method first proposed by Coveyou and McPherson in 1965, which is based on the finite Fourier transform. The mathematics underlying this test, the Spectral Test of an RNG, are beyond the scope of this article, but they are well described by Knuth. The Fourier transform itself is a mathematical technique for detecting and describing repeating patterns in a set of data.

To compute the values of the V's, we used Knuth's Algorithm S, which requires high precision Integer arithmetic. Apple's Pascal provides Long Integer type variables, which allowed us the Integer precision we needed. (We do not list this program because it is a direct implementation of Knuth's algorithm S). The algorithm takes only a and m as input -- the value of c is irrelevant. It quickly determines the values of the V's for the output of the generator across its entire period.

This is so spectacular that we want to say it again. The statistics V2, V3, and so on, which take only minutes to calculate, take into account the ordering of every one of the 1,099,511,627,776 different values the random number generator produces. (!)

Until the theorems behind this amazingly powerful algorithm were proved, testing of random number generators was done by examining a relatively "small" subset of the sequence the generator produced. "Small" here means maybe a million numbers. On an Apple, this type of testing can take months of computer time. (We report some subsequence tests below, and they took days. Tests of other generators not discussed here actually did take months). One of the reasons that old generators are so poor, relatively speaking, is that it took so long to test one. Testing of replacements was a tedious and very expensive business.

To choose the multipliers for the three generators we present here, we computed V values for just over 30,000 different values of a. (A life's work for at least 100 long-lived Apples if they were all tested in the old ways, and this only took two weeks). We stopped when three suitable values of a were found.

The values of the V's tell us what the largest distance is between a pair of lines, planes or hyperplanes in a subsequence graph of the entire period. These values depend on the period of the generator: the larger m is, the larger V can be. These numbers can get so large (see Table 1) that it's very hard to tell whether a value of V is good or not. For any given period, there is a best possible value for each of the V's. The easiest way to tell how a given V value rates is to convert to a different number (call it U), that takes the period of the generator into account. Knuth gives formulas for converting from V2 to U2, V3 to U3, ..., V6 to U6. The Spectral test is usually defined in terms of the U values. If U is greater than 0.10, the generator "passes" the test. According to Knuth, every generator known to be bad fails the test at this level. He defines a "pass with flying colors" as a value of U greater than 1.0.

The Spectral test is the most powerful test known of random number generators. The U and V values should be part of the documentation of any RNG. We list the values of the three generators presented here in Table 1. Our smallest U is 2.37.

For comparison, the U values of RANDU, a very common RNG on 32-bit mainframes, are 3.14 (U2), less than 0.0001 (U3), less than 0.001 (U4), less than 0.01 (U5) and 0.02 (U6).

Table 1
Results of the Spectral Tests

Generator	X USR(1)	Y USR(0)	Z USR(-1)
Multiplier	27182819621	8413453205	31415938565
Additive Constant	3	99991	26407
V2	982974962600	1112748837514	908473954394
V3	72937326	103184754	79566866
V4	1023550	805970	1036504
V5	58786	60670	59710
V6	9916	8142	11636
U2	2.81	3.18	2.60
U3	2.37	3.99	2.70
U4	4.70	2.91	4.82
U5	4.01	4.34	4.17
U6	4.58	2.54	7.40

Knuth (pp. 102-104) provides a table of U and V values for many mainframe generators. Most (fortunately) are better than RANDU. Some are better than the three we are presenting here, but not many of them.

The problem with many of the older generators is that they were speed-optimized. A full period is obtained from any generator whose c is odd and whose a is any even power of 2, plus 1. These are not the only full-period multipliers (far from it), but if you choose a so that it is a power of 2, plus 1, all that you have to do in the multiplication is to shift the accumulator a few times (the multiplication degenerates into a simple set of shifts), and then add.

As an example of a fast generator, if you choose a = 2^8+1 and a 32-bit generator, as was recommended for the Apple not too long ago by someone else, you don't even need to shift anything. Add the lowest byte to the second lowest. Add the (8-bit plus carry) sum of these to the third lowest byte. Add the (8-bit plus carry) sum of these to the high byte and you are done. This is short, sweet, elegant, very fast, it passes some of the sub-sequence tests, but it fares badly on the Spectral test and it would probably be inadequate for many applications.

We aren't going to say who suggested this generator, or in what magazine, because it could needlessly embarrass an author who doesn't deserve to be embarrassed. He consulted a standard, fairly recent (1971), and well written text on random numbers (Newman & Odell's, The Generation of Random Variates), followed their recommendations, and conducted their tests. Unfortunately, the importance of the lines and planes problem wasn't widely enough or fully enough realized in 1971, and the full-period tests, many of which had not yet been developed or polished, were not widely enough appreciated. Newman and Odell's otherwise very good summary of generation techniques and applications of random numbers made virtually no mention of full-period results. Their recommendations favored multipliers with few bits set, such as 2^8+1 or 2^8+3. Similarly, Abramowitz and Stegun's numerical bible (also known as the Handbook of Mathematical Functions, 1964) recommends generators of the power of 2 plus 1 type. Finally, and in another book deserving a home on any programmer's bookshelf, Carnahan, Luther and Wilkes' Applied Numerical Methods (1969) makes much the same recommendation.

The very fast generators, with few bits set to allow jazzed-up multiplication routines, have generally fared badly when subjected to the Spectral test. RANDU, for example, used a multiplier of $2^{16}+3$. The problem seems to be that so few bits are set, and so few operations are thus performed on the number, that the number's digits are not sufficiently scrambled each time. In the 1950's and early 1960's, generators of this type were considered ideal, rather than poor. They passed many of the simpler tests of randomness. And, critical for large simulations then, fast meant (relatively) cheap. (We keep talking about cost. Here's an illustration that makes the point. In 1978-79, Kaner and John C. Lyons conducted a moderately large simulation of the behavior of the Kolmogorov-Smirnov and related statistics, using three PDP-11 lab minicomputer. Some tests of the validity of their work required greater numerical precision than was easily obtained on the PDP's, so they also did some work on a CDC 6400 mainframe. Out of curiosity, they ran benchmark tests to determine how much the simulation would have cost if all of the work had been done on the CDC. It would have cost over $100,000, or more than enough, at that time, to but three well equipped PDP-11's).

The recognition that tests of sequential patterning are more important than tests of frequency (discussed below) for generators that produce all possible numbers between 0 and m, and the discovery of fast techniques to search for patterns across the entire period, have caused something of a revolution in the way RNG's are created and tested. Almost all of this has taken place over the last 20 years, and much more has yet to come.

Readers familiar with statistical techniques may have grumbled, by this point, that there were tests of sequential patterning long before the Spectral test. We will mention the results of a few of these below, but one of them, the Serial Test, is relevant here.

Suppose that you split the range of fractions generated (R(N|m) into 10 equal subranges, 0 to .10, .10 to .20, .20 to .30 and so on. If you generate a sample of 10,000 numbers, you can count how many fall in each subrange. A random source would produce about 1000 for each, and this can be compared to the number that the RNG produces. This simple test, of the frequency of single numbers (rather than of pairs or triples), is called the Chi-Square Test. Similarly, you can examine the pairs, (R[I], R[I1]). From a sample of 10,000, you should obtain approximately 100 of each type of pair. That is, in about 100 cases, both R[I] and R[I1] should be between 0 and .10. In another 100 cases, R[I] should be less than .10 while R[I1] is between .10 and .20, and so on. There are 1000 types of triples (R[I], R[I1], R[I2]) and on average we'd expect to obtain 10 of each. The traditional test used to examine the difference between the actual number of each number, pair, triple, etc. and the number that we should obtain on average is called the Serial Test. There are a number of versions. We prefer Good's, developed in 1953. (Knuth presents a different one that is also popular.)

The Serial test is a subsequence test. You take a sample of the numbers produced by the generators (we used the first 850,000 from each in our tests, for example). If you didn't mind tying up your Apple for a few years you could test the entire output of the RNG (all trillion-plus numbers), obtaining a full-period test the hard way. For such a large sample, this test is known to be extremely sensitive to deviations from randomness.

Over the last ten years, Neiderreiter has proved a very important set of theorems about the relationship between the Spectral test and the full-period Serial test (see Knuth for references and details). In short, any RNG that passes a full period Spectral test will also pass a full period Serial test. By using the Spectral test to determine the three values of a we ensured that the RNGs would pass both tests.

We have now settled on values for m and a. How do we decide what c should be? The additive constant in the generator makes no difference for the Spectral test, but it does influence the value of another traditional test of ordering, the Serial Correlation Test. You can think of the serial correlation, lag K, as a measure of the degree to which the relationship between R[I] and R[IK] can be described as linear. A value of 0 indicates that there is no linear relationship between the random number produced now by the RNG and the value that it will produce K calls from now. A value of ± 1 indicates a perfect linear relationship, and an atrocious RNG.

Knuth's Theorem K gives a method to establish upper and lower bounds on the correlation, across the entire period. We applied it to test several additive constants, for each of the RNGs, for serial correlations lag 1 through 20 (again in a Pascal program not listed here that followed Knuth directly). There were thus 60 correlations, 20 for each generator. For the values of c chosen, the largest correlation lies somewhere between -0.00000001135 and 0.00000000569. The second worst case lies between -0.00000000038 and 0.00000000072. We don't know the exact values, just the upper and lower bonds on the correlations, but whatever they are they are pretty close to zero, which is where they should be.

In summary, the modulus value of 2^{40} resulted from a compromise between considerations of speed and space on the one had, and of period length and tiny value representation on the other. The critical full period tests from here were tests of sequential relationship. Equal frequency is, of course, a major criterion of randomness, but this entered into our parameter selection only insofar as values of a and c that would not guarantee equal frequency were rejected automatically. The parameter selection was determined, for each generator, by performance on the major full period tests of sequential relationship.

Empirical Tests of the RNGs

Full period tests only tell us about the performance of the RNG across the entire period. They do not guarantee that the sub-sequences will be good. It could be that a strong trend in the first 100,000 values will be counterbalanced by a reverse trend in the next 100,000, and so on. Since no application that we know of would use the full trillion number period, the only way to be confident about quality for actual use is to examine the RNG's sub-sequence behavior.

To do this, we ran a number of standard statistical tests on the output of each generator, examining the output in batches (sub-sequences) of 1,000 to 10,000. For each test, sampling started at the (same) starting values of the generators. Many users will only need these first few hundred thousand numbers, so these should be the ones most carefully examined.

1) Serial Tests

We described these in the final discussion of the Spectral test, above. Samples of 10,000 numbers were tested for simple frequency (number of R[I]'s < .10, between .10 and .20, etc.) and for clustering of pairs and triples. Eighty-five batches were examined and the 85 results, for each test and each generator, were compared to the distribution of results we would expect from a random source, using the Kolmogorov-Smirnov Test. All three generators passed the simple frequency ($p > .10$) and triples ($p > .20$) tests. The generators listed as X and Y in Table 1 passed the doublets test, but Z did not. The problem with Z, which we will see again later, is that it does too well on these tests.

If you test a truly random source many times, it will sometimes fail a test of randomness and it will sometimes only marginally pass it. Not often, but sometimes, and we can calculate how often theoretically. Z's performance was sometimes poor, but not often enough to mimic a random source ($.05 > p > .02$). Since nothing can be "more random"than a "truly random" source, this must be a flaw in Z.

It should be realized, though, that these tests are very sensitive to minor deviations from random source behavior when such huge (850,000) sample sizes are involved. Z does not perform ideally, but its performance is far from bad.

2) Frequency Tests

Equal frequency over the full period is guaranteed in a full period generator: one and only one of each number is produced each time through the trillion number series. But the fact that all possible numbers will eventually appear is no guarantee that they will come in a reasonable order. It all too often turns out that an RNG yields too few, then later too many small (large, whatever) numbers. We described the Chi-Square test of frequency as a special case of the Serial test. (Historically, the Serial was an extension of Chi-Square). A different test requires no grouping of the numbers and it is often more sensitive to departures from equal frequency than Chi-Square. This test, the Kolmogorov-Smirnov test (KS test for short), compares the proportion of numbers generated that are less than any given number (across all numbers between 0 and 1) with the proportion that we'd expect from a random source.

A hundred such tests, for each generator, were run on batches of 1,000 numbers, and the KS test was then used to compare the distribution of KS values from these 100 tests with the distribution a truly random source would give. X and Y passed ($p > .20$). Z failed, even though it had already passed Chi-Square. The problem with Z, as before, was that its test performance was too good, too often ($.05 > p > .02$). This is a most unusual problem for an RNG, but searching techniques for "terrific" generators, like the search we performed across 30,000 potential generators, are becoming commonplace, so we can expect this to arise more often.

3) Runs Tests

A run up is a succession of increasing numbers (eg. .1, .2, .35, .36) which ends when the next number generated is lower than the last. A run down is similar. In this case, successive numbers get smaller. The number of increasing or decreasing numbers in a run is the length of the run. Tests of how many runs, and how many runs of each length, are further tests of sequential trend in the RNG. Both types of tests were run, for each generator, on a sample of 50,000 numbers. All generators passed them handily.

4) Other Tests

We developed these generators two years ago (summer, 1981) and have used them often since. Kaner has mainly used them to simulate logistic, triangular, normal, and geometric distributions, and the behavior of various functions of variables having these distributions (such as the kurtosis of weighted sums of a logistic plus a triangular plus a geometric, which is an important variable in a theory of time perception that he works with). Z was never used in these simulations. X and Y performed quite adequately. Numerous comparisons of theoretically predictable values with the simulation results were made along the way, and none of the comparisons suggested any problems with the RNGs.

Vokey has conducted simulations involving binomial, t, F, and various other distributions of common hypothesis testing statistics, and of multinomial and hypergeometric distributions and functions of these involved in theories of choice and category learning. X and Y have performed well consistently. Z has performed strangely: extreme values of complex statistics are not as likely as they should be with Z.

In sum, X and Y have passed all tests, theoretical (full period) and empirical (sub-sequence). Z's sub-sequence behavior has been less good (i.e., too good), and the more of it that we see, the more hesitant we are to use it again as a "stand-alone" RNG. This does not mean it's useless, as we shall see below.

X and Y appear sufficiently random for most needs, and they have performed empirically beyond our hopes for them. But they are not perfect. We have minimized the lines and planes problem, but we have not gotten rid of it. For very precise simulations, especially of events correlated over time, this is not go enough. However, if more than one RNG is available (which is why we provide three), we can do much better than we have done so far.

Combination of RNGs

The graph of the last number generated, R[N], against the number generated this time, R[N 1], shows a family of parallel lines when all pairs (R[N], R[N1]) are plotted (as illustrated in Figure 1). This is the parallel lines problem. If our goal is to break down this linear structure, as we must do to mimic the random structure produced by a truly random source, why not just randomly rearrange the order of the numbers generated by the RNG, as it produces them? This is George Marsaglia's insight, and in practice it works out very well.

Here's an example of the procedure for wiping out the lines and planes patterns. Generate 100 values from X and store them in a matrix, say XRAN[I]. Now sample a value from generator Y and use this value to determine which value you'll choose this time from XRAN, i.e. set RANDOM XRAN[Y * 100]. Replace the sampled value of XRAN with a new value from generator X (XRAN[Y * 100] USR (1)) and you're done.

A sequence of numbers, RANDOM [I], will have the same good sub-sequence frequency properties as X does, but the last remnants of sequential patterning from X typically disapper. Knuth gives examples of quite poor generators which perform surprisingly well when combined in this way. All combinations that we've examined in X, Y and Z have been good, but we recommend that Z be restricted to the role of selection generator (the role played by Y in the example above) due to its too equal sub-sequence frequencies. We see no problem in using Z as a selection generator. Some people would argue that Z might be a better selection generator than X or Y. We're not sure.

A second approach is to sample from X, then Y, then Z, in turn. This triples the period and it can destroy the lower-dimensional patterns (the lines), but it will not do for all generators combined in this way. In fact, Lewis' Multi-RNG Theorem (pp. 18-19) states that if any multiplier in a bank of equal period RNG's is near sq.rt. m , the problem will return with a vengeance. (A sad result because the old generators were often chosen to be near sq.rt. m deliberately, and the older texts recommend this heartily). We restricted selection of multipliers for X, Y and Z to values far from sq.rt. m = 1,048,576 in order to allow this form of generator combination. For these generators, according to Lewis, the technique should be very effective.

The last approach that we'll mention is to use one generator (Z) to decide which of the other two will be sampled from this time. This only doubles the period of the resulting generator (if you need 3 trillion numbers, use a different RNG), but it does randomize the order of sampling from the generators, which is not done above.

It seems probably important for each solution above that the different generators' outputs be unrelated. Otherwise, replacing a value of X with one from Y, for example might make little difference. Our final test of the generators involved computing the correlation (measure of linear relationship) between X and Y, Z and Y and Z. A hundred correlations were taken, on samples of 1,000 numbers per generator. All were reasonably low. The averages were 0.0003 for X and Y, 0.0038 for X and Z, and -0.0017 for Y and Z, which should be low enough to allow combinations.

Using the RNG

Once you have entered the RNG program into your Apple (below), you access it via the USR function. A statement of the form

```
RAN USR(SELECT)
```

in either immediate or deferred mode, will put a random number into the variable RAN. SELECT must be a Real or Integer number, variable or expression. If it is less than 0, the output will be from Z. If SELECT is 0, the output is from Y. If SELECT is larger than 0, output is from X. If SELECT is a String, output is "?SYNTAX ERROR".

Some of the locations of the program hold the last values generated by each RNG. Unless you are debugging a program and want the same number sequence again and again (in which case, see below), you should never use the same random numbers twice. If you never have to reload the program, this is taken care of automatically. However, if you must reload the program, it will start from the initial values of 3, 99991 and 26407. It is easy to avoid this problem by always updating your copy of the program on the disk. At the end of every program that uses an RNG, we PEEK the contents of decimal locations 768 to 969 (the entire program) and save them on the disk. (Equivalently, BSAVE RNG, A$300, L$C9). At the start of every RNG-using program, we BLOAD the program from the disk. This ensures that we always start with the next random number in the sequence.

To obtain a standard sequence instead, keep another copy of the RNG program, and deposit it into core as needed, but never update it. This downloads the same values every time, yielding the same sequence every time.

The RNG Program

The program starts by determining which generator is requested, and does so by calling Applesoft's internal SIGN subroutine. Variable LOOKUP holds the offset value, determined from the sign variable in USR(), which, when added to ADDBAS, yields the final location of XADD (LOOKUP = $22), YADD($13) or ZADD ($04). These are the additive constants, c, of the generators and the final locations are used because our DO loops are most conveniently done as 4 DOWNTO 0.

LOOKUP added to MULBAS (multiplier base address) yields XMULT, YMULT or ZMULT, the values of a.

LOOKUP + LSTBAS points to the last value generated from X, Y or Z, XRAN, YRAN or ZRAN, i.e. to the R[N] of R[N + 1] = (aR[N] + c) mod m.

By loading the appropriate value of LOOKUP into the computer's Y register, we access the appropriate RNG. To avoid constantly worrying about which generator we are dealing with, we store the values in a set of standard locations (freeing register Y for other uses).

NEWRAN will hold the new value generated. By depositing c into it directly, we perform the addition of c automatically. MULT and OLDRAN hold a and R[N]. This is the function of the program segment labelled TRNSFR.

The next section of the program thoroughly confuses readers unfamiliar with modular arithmetic. Remember that the value of A mod B is the value of the remainder of the division of A by B. The quotient itself is irrelevant. Since 2^{40} will divide evenly into any multiple of 2^{40}, any number greater than 5 bytes in length reduces to the least significant 5 bytes (40 bits) directly. Every bit more significant than the 40th (or 39th if you number from 0) is an even multiple of 2^{40} so it cannot enter into the remainder of the division. Combine this with the fact that

```
(AB) mod C (A mod C B mod C) mod C
```

and you will see that we never need any bits past the 40th. Thus we never store them. The multiplication segment calculates the least significant 40 binary digits and quits. The additions always ignore the carry from the sum of the highest bytes.

The multiplication algorithm is the same as the one we all learned in elementary school. Here is an example of standard multiplication:

```
    12345  OLDRAN
  X 11111  MULT
    12345
   1 2345
  12 345
 123 45
 1234 5
 1371 65295  PROD
```

To multiply OLDRAN by MULT, we multiply by the least significant digit of MULT, shift OLDRAN left by one, multiply the next least significant digit, shift again, etc. If MULT held a zero at any point, we would shift OLDRAN for the next digit of MULT, but not add anything to the answer, PROD. In the RNG program, we do it the same way, with register Y keeping track of the bits of the byte of MULT by which we are currently multiplying. Index X and BYTCNT keep track of MULT's byte. The difference between our algorithm and the traditional one is indicated by the vertical line in the example. We need no digits past the line, so we never calculate anything past it.

The third section of code, MOVRAN, is executed after the multiplication and the addition are done. We now have R[N + 1], which we store as is in XRAN, YRAN or ZRAN. Which generator it goes back to is kept track of by XYORZ. R[N + 1] is always stored as a fixed-point integer, because all computations are and always should be fixed-point integer for the modulo operation to work. Why store a normalized value and have to decode it each time?

(Note Bene: We now know of three cases in which programmers have "improved" on congruential generators by doing floating point computations rather than fixed integer computations. More precision is better, right? Wrong! Not here! The theorems we reference above all assume fixed-point integer arithmetic, with truncation not rounding. The computational errors involved in integer arithmetic are part of the algorithm. Maybe floating point calculations will be good for some generators, but this is uncharted territory. In our experience, this "improvement" has always led to a bad generator).

NRMLIZ puts NEWRAN into the floating-point format that Applesoft expects. To convert fixed-point to floating-point, left-rotate the number until its most significant digit (the first set bit) is the leading bit of the number. As long as we keep track of the number of rotations performed (held in register Y), we could convert back to fixed point easily if we wanted to. Floating-point format, which stores an exponent (reflecting the number of rotations) along with the normalized digits, allows a wider range of values to be stored to high precision than does fixed-point format.

Once normalization is done, we either branch to BITSET if a set bit (most significant digit) was found, or fall through to it if R[N] = 0. At BITSET we first load a 0 into a byte reserved to hold the number's sign (making it positive), then convert to Apple's convention for storing exponents. In this format, if the exponent is $80, no rotations were required and the number lies between

0.999999999 and 0.50. If the exponent is $70, one rotation was done and the number lies between 0.5 and 0.25, and so on.

To store our integer as a fraction, we need only load the exponent with a value no greater than $80, and less than $80 by the number of rotations needed to get the top set bit. This is exactly equivalent to dividing R[N+1] by 2^{40}, except where R[N+1] is 0. In this case, the program returns 2^{41} instead of zero. This is close to zero but it removes the chance of a "?DIVISION BY ZERO" error if you divide by a random value. This is probably academic anyway, as the starting values ensure that 0 will be the 1,099,511,627,776th number generated by each RNG.

Below are the parameters for the Apple, Commodore, and essentially any other 6502 machine that uses a Microsoft BASIC. See Editor's Note, Page 34.

APPLE - to set up the USR function - POKE 10,76: POKE 11,61: POKE 12,3 (These would be altered if you are starting the program elsewhere in Memory.)

```
MULTMP   EQU  $9D
RANEXP   EQU  $9D
NEWRAN   EQU  $9E
BYTCNT   EQU  $AC
SIGN     EQU  $EB82
```

COMMODORE - to set up the USR function - POKE 785 and 786 with address of RNG Subroutine.

```
MULTMP   EQU $61
RANEXP   EQU $61
NEWRAN   EQU $62
BYTCNT   EQU $Ø2
SIGN - see below
FEXP     EQU $61
FSGN    EQU FEXP+5
```

The following is the Applesoft sign routine converted to a form for the Commodore. In line 0344 of the main program there is a JSR to SIGN. The location $EB82 is where the sign routine is located in the Apple. For the Commodore you can place the SIGN subroutine anywhere as it is completely relocatable.

Floating Point Exponent - FEXP
Floating Point Sign - FSIGN

```
C64:      FEXP   EQU  $61
         FSIGN   EQU  $66
APPLE:    FEXP   EQU  $9D
         FSIGN   EQU  $A2
A5 61 SIGN LDA FEXP
FØ Ø9      BEQ RTN
A5 66      LDA FSGN
2A         ROL A
A9 FF      LDA $FF
BØ Ø2      BCS RTN
A9 Ø1      LDA $Ø1
6Ø     RTN  RTS
```

Listing 1

```
Ø3ØØ                       ORG $3ØØ
Ø3ØØ              ************************************
Ø3ØØ              *                                  *
Ø3ØØ              *  A BETTER RANDOM NUMBER GENERATOR *
Ø3ØØ              *         FOR APPLESOFT            *
Ø3ØØ              *                                  *
Ø3ØØ              *         COPYRIGHT 1984           *
Ø3ØØ              *      THE COMPUTERIST INC.        *
Ø3ØØ              *      ALL RIGHTS RESERVED         *
Ø3ØØ              *                                  *
Ø3ØØ              ************************************
                 ;
                 ; TO USE THE RNG SUBROUTINE, YOU MUST
                 ; SET UP THE USR FUNCTION.
                 ; SEE EDITORIAL NOTE
                 ;
                 ;  LOAD IN PARAMETERS FOR THE RNG'S
                 ;
                 ; Z: RAN(31415938565*OLD246Ø7)MOD2Ø
                 ;
Ø3ØØ ØØ ØØ ØØ     ZADD     BYT $ØØ,$ØØ,$ØØ,$67,$27
Ø3Ø5 Ø7 5Ø 89     ZMULT    BYT $Ø7,$5Ø,$89,$2E,$Ø5
Ø3ØA ØØ ØØ ØØ     ZRAN     BYT $ØØ,$ØØ,$ØØ,$ØØ,$ØØ
                 ; Y: RAN(84134532Ø5*OLD99991)MOD2Ø
                 ;
Ø3ØF ØØ ØØ Ø1     YADD     BYT $ØØ,$ØØ,$Ø1,$86,$97
Ø314 Ø1 F5 7B     YMULT    BYT $Ø1,$F5,$7B,$1B,$95
Ø319 ØØ ØØ ØØ     YRAN     BYT $ØØ,$ØØ,$ØØ,$ØØ,$ØØ
                 ; X: RAN  (27182819621*OLD3)MOD2Ø
                 ;
Ø31E ØØ ØØ ØØ     XADD     BYT $ØØ,$ØØ,$ØØ,$ØØ,$Ø3
Ø323 Ø6 54 38     XMULT    BYT $Ø6,$54,$38,$E9,$25
Ø328 ØØ ØØ ØØ     XRAN     BYT $ØØ,$ØØ,$ØØ,$ØØ,$ØØ
                 ; ADD LOOKUP TO BASE LOCS FOR
                 ; PARAMETER ADDRESSES FOR CURRENT RNG.
Ø32D Ø4 13 22     LOOKUP   BYT $Ø4,$13,$22      ; Z, Y, X
                 ;
Ø33Ø ØØ           XYORZ    BYT $ØØ           ; WHICH GENERATOR
Ø331 ØØ           YTEMP    BYT $ØØ           ; Y-REG ON ENTRY
Ø332 ØØ           XTEMP    BYT $ØØ           ; X-REG ON ENTRY
Ø333 ØØ ØØ ØØ     MULT     BYT $ØØ,$ØØ,$ØØ,$ØØ,$ØØ
Ø338 ØØ ØØ ØØ     OLDRAN   BYT $ØØ,$ØØ,$ØØ,$ØØ,$ØØ
                 ;
Ø33D Ø8           RNG      PHP               ; SAVE EVERYTHING
Ø33E 8E 32 Ø3              STX XTEMP
Ø341 8C 31 Ø3              STY YTEMP
Ø344 2Ø 82 EB              JSR SIGN          ; SEE EDITOR'S NOTE FOR
                                             ; SIGN ROUTINE
                                             ; FAC HOLDS S OF USR(S)
                 ;                             PUT FF IN A IF SØ,
                 ;                             PUT Ø IF Ø, 1 IF SØ
Ø347 AA                    TAX               ; FROM THIS
Ø348 E8                    INX               ; DECIDE WHICH RNG
Ø349 BC 2D Ø3              LDY LOOKUP,X      ; VIA LOOKUP TABLE AND
Ø34C 8C 3Ø Ø3              STY XYORZ         ; SAVE IT FOR LATER
                 ;
                 ; NOW THAT WE KNOW WHICH GENERATOR, MOVE
                 ; ITS CONSTANTS TO THE TEMP LOCS.
                 ;
Ø34F A2 Ø4                 LDX $Ø4          ; LOOP TO TRANSFER
Ø351 B9 ØØ Ø3     TRNSFR   LDA ADDBAS,Y     ; RNG'S VALS TO
                 ;                             STANDARD LOCS, I.E.
Ø354 95 9E                 STA NEWRAN,X     ; ADD CONST TO NEWRAN,
Ø356 B9 Ø5 Ø3              LDA MULBAS,Y     ; MULT CONST
Ø359 9D 33 Ø3              STA MULT,X       ; TO MULT,
Ø35C B9 ØA Ø3              LDA LSTBAS,Y     ; LAST RND VAL FROM
Ø35F 9D 38 Ø3              STA OLDRAN,X     ; THIS RNG TO OLDRAN
Ø362 88                    DEY
Ø363 CA                    DEX              ; 5 BYTES DONE
```

```
Ø364 1Ø EB                   BPL TRNSFR    ; IF NO, DO NEXT
                  ;                          IF YES, MULTIPLY.
Ø366 A2 Ø4                   LDX $Ø4      ; INDEX  OF BYTES
Ø368 86 AC                   STX BYTCNT    ; KEEP TRACK OF  BYTES
                  ;                          DEALT WITH SO FAR
Ø36A BD 33 Ø3    NXTBYT      LDA MULT,X    ; LEAST SIGNIF BYTE
Ø36D 85 9D                   STA MULTMP
Ø36F AØ Ø7                   LDY $Ø7      ; COUNT  BITS
Ø371 46 9D       MULPLY      LSR MULTMP    ; GET LEAST SIG BIT.
Ø373 9Ø ØC                   BCC SHIFT     ; BITØ DON'T ADD.
Ø375 18                      CLC           ; BIT SET, SO ADD
Ø376 BD 38 Ø3    ADD         LDA OLDRAN,X  ; OLDRAN TO NEWRAN.
Ø379 75 9E                   ADC NEWRAN,X
Ø37B 95 9E                   STA NEWRAN,X
Ø37D CA                      DEX           ; ALL BYTES DONE
Ø37E 1Ø F6                   BPL ADD       ; NO ADD NEXT
Ø38Ø 18                      CLC           ; YES, SO PREPARE TO
                  ;                          SHIFT OLDRAN (IE
                  ;                          MULT * 2). DROP LAST
                  ;                          CARRY AS IT IS
                  ;                          Ø MOD2Ø ANYWAY.
Ø381 A6 AC       SHIFT       LDX BYTCNT    ;  BYTES TO SHIFT
Ø383 3E 38 Ø3    SHFTIT      ROL OLDRAN,X
Ø386 CA                      DEX           ; BYTE LEFT
Ø387 1Ø FA                   BPL SHFTIT    ; YES, SHIFT IT.
Ø389 A6 AC                   LDX BYTCNT    ; RECOVER  BYTES.
Ø38B 88                      DEY           ; MORE BITS LEFT
                  ;                           IN THIS BYTE
Ø38C 1Ø E3                   BPL MULPLY    ; YES, MULT BY NEXT.
Ø38E C6 AC                   DEC BYTCNT    ; NO, DONE A BYTE.
Ø39Ø A6 AC                   LDX BYTCNT    ; ANY BYTES LEFT
Ø392 1Ø D6                   BPL NXTBYT    ; YES MULT BY IT.
                  ;
Ø394 AC 3Ø Ø3                LDY XYORZ     ; DONE. PUT THE
Ø397 A2 Ø4                   LDX $Ø4      ; NEW RND INTO THE
; RESPECTIVE RNG'S
Ø39B 99 ØA Ø3                STA LSTBAS,Y  ; LAST RAN STORAGE.
Ø39E 88                      DEY
Ø39F CA           ;          DEX          ; MORE TO MOVE
Ø3AØ 1Ø F7        ; DONE. NOW TO NORMALIZE FAC, ALIAS NEWRAN.
                  ;
Ø3A2 AØ 28                   LDY $2      ; $28 (4Ø) BITS IN FAC.
Ø3A4 A5 9E       NRMLIZ      LDA NEWRAN    ; FIND HIGHEST SET.
Ø3A6 2A                      ROL           ;  SIGNIFICANT
                  ;                          28 -  NOT SET
Ø3A7 BØ ØE                   BCS BITSET    ; LEAVE WHEN TOP
                                             BIT FOUND
Ø3A9 26 A2                   ROL NEWRAN4  ; NOT FOUND YET, SO
Ø3AB 26 A1                   ROL NEWRAN3  ; GET RID OF THE Ø
Ø3AD 26 AØ                   ROL NEWRAN2  ; BIT AT THE TOP.
Ø3AF 26 9F                   ROL NEWRAN1  ; Y WILL KEEP TRACK
Ø3B1 26 9E                   ROL NEWRAN    ; OF  OF BITS LEFT.
Ø3B3 88                      DEY           ; ANY LEFT
Ø3B4 DØ EE                   BNE NRMLIZ    ; YES, KEEP LOOKING
                  ;                          NO, ALL DONE.
Ø3B6 88                      DEY           ; PROTECT AGAINST
                  ;                          DIVIDE BY Ø.
Ø3B7 A9 ØØ       BITSET      LDA $ØØ      ; PUT Ø IN FAC'S
Ø3B9 85 A2                   STA NEWRAN4  ; SIGN BYTE.
Ø3BB 98                      TYA           ; GET  SIG BITS
Ø3BC 18                      CLC           ; PUT IN FAC'S $8Ø
Ø3BD 69 58                   ADC $58      ; FORMAT: $58$28$8Ø.
Ø3BF 85 9D                   STA RANEXP    ; PUT IN EXPONENT
                  ;                          BYTE AND DONE.
Ø3C1 AC 31 Ø3                LDY YTEMP     ; SO, UNSAVE
Ø3C4 AE 32 Ø3                LDX XTEMP     ; EVERYTHING
Ø3C7 28                      PLP           ; AND
Ø3C8 6Ø                      RTS           ; SAY GOODBYE.
Ø3C9                         END
```

References

Abramowitz, M & Stegun, I. A., Handbook of Mathematical Functions, National Bureau of Standards, 1964 (Reprinted with Corrections by Dover Press, 1972).

Carnahan, B., Luther, H. A., & Wilkes, J.O., Applied Numerical Methods, John Wiley & Sons, 1969.

Good, I. J., The serial test for sampling numbers and other tests for randomness, Proceedings of the Cambridge Philosophical Society, 1953, Vol. 49, 276-284.

Kaner, H. C. & Lyonss, J.C., Tables and Power Comparisons for Different Versions of the Kolmogorov-Smirnov and Schuster Statistics, Technical Report No. 67, Department of Psychology, McMaster University, 1979.

Kendall, M. G. and Stuart, A., The Advanced Theory of Statistics, Vol. 2: Inference and Relationship (third edition, 1973), Vol. 3: Design and Analysis, and Time Series (third edition, 1975), Hafner Press.

Knuth, D. E., The Art of Computer Programming, Vol. 2, Seminumerical Algorithms. Addison:Wesley, 1981 (Second Edition).

Lewis, T. G., Distribution Sampling for Computer Simulation. Lexington Books, 1975.

Marsaglia, G., Random Number Generation. In A. Ralston's Encyclopedia of Computer Science, Van Nostrand Reinhold, 1976, 1192-1197.

Newman, T. G. & Odell, P., L., The Generation of Random Variates, Hafner Publishing Co., 1971.

Editor's Note: When adapting this random number generator subroutine we found it to be essentially free from machine specific code. The two places the code differs are in the use of the USR function which accesses the program from a BASIC file, and in the use of floating point notation, in particular the APPLESOFT Sign routine. After examining the available documentation the USR function for the Apple and Commodore we found it wasn't clear as to how parameters were passed. The locations used were different, but this was expected. The question was how the floating point notation operated. To solve this problem we wrote a small program (see below) which allowed us to display the floating point accumulator, PEEKing the locations where the exponent, mantissa and sign were stored in each computer. If they were stored differently, then further modifications would have to be made. Happily our little program proved that they store the parameters in the same form. Now for the bad news - we found the Atari didn't use floating point notation in its USR function. This, combined with a different convention for storing floating point notation (combining the exponent and the sign), made easy adaption of this program impossible. Certainly if those readers with Atari's wish to meet the challenge it can be done. Bear in mind the different USR function and the use of floating point notation in the RNG subroutine, and how it would have to be changed to accomodate the Atari's conventions.

```
10 REM PROGRAM TO DISPLAY FLOATING POINT
   ACCUMULATOR
20 UV10 : REM USER VECTOR
30 PN3 : REM PAGE NUMBER
40 FA157 : REM FLOATING POINT ACCUMULATOR
50 MPPN*256 : REM MEMORY PAGE
60 POKE UV,76:POKE UV1,00:POKE UV2,PN
100 MPPN*256 : REM MEMORY PAGE
110 POKE UV,76:POKE UV1,00:POKE UV2,PN
120 MVMP:SV128:I0
130 POKE MV,165:POKE MV1,FAI
140 POKE MV2,141:POKE MV3,SV:POKE MV4,PN
150 MVMV5:SVSV1:II1
160 IF I6 THEN 130
170 POKE MV,96
200 INPUT VALUE ;A
210 BUSR(A):PRINT VAL ;B
220 PRINT EXP ;PEEK(MP128)
230 PRINT MSB ;PEEK(MP129)
240 PRINT     ;PEEK(MP130)
250 PRINT     ;PEEK(MP131)
260 PRINT LSB ;PEEK(MP132)
270 PRINT SIGN;PEEK(MP133)
280 GOTO 200
```

CONTROL

by Mitchell Esformes

Test your assembly code for efficiency, or adapt the program for statistic, step/trace debugging and more.

CONTROL is a machine language program that runs your machine language/assembly level program instruction-by-instruction and allows you to control its operation and/or collect statistics about your program. This could be used for as simple an application as counting cycles in a program (details shown in this example), as a step/trace function with disassembly of each instruction, as a sophisticated debugging tool that traps on specified instructions and/or memory locations, and so forth.

The program works by setting up a pseudo program counter, fetching and evaluating each instruction from the program under control, performing any special operations that you define, and then executing the instruction. It is written in such a way that you can easily add your own processing routines. The demonstration process shown here simply counts the number of machine cycles used by a program. While this can be useful in developing optimally efficient code, it is only a hint of what can be done with this technique. CONTROL will run on any 6502 microcomputer. The only requirements are four page zero locations and about 1K of program space.

Program Description

EQUATES: These are the addresses of locations used by the program for its program counter, table pointers and for saving the 6502 registers. PCTR is a two byte page zero vector which contains the pseudo program counter; TEMPLO and TEMPHI are a page zero pair of bytes used for vectoring to the CONTROL tables. The 6502 registers are saved in ACC (A reg), XREG, YREG, STREG (status) and STKPTR (stack pointer). These do not have to be on page zero. TALLY is an eight byte table used to hold the cycle count in this particular example. If you design some other function for the CONTROL program, then this will not be needed.

START puts the address of a location containing a BRK command, BREAK, on the stack to be used to halt the program when an RTI is encounted in the test program, sets the status to 0 to enable interrupts, and clears the TALLY counter. If you are not counting cycles, the TALLY counter does not have to be cleared.

FETCH is the beginning of the main processing loop. It picks up the first byte of the current instruction, the OPCODE, and converts it to the range $00 to $0F to speed up the table lookup. The table INST1 is searched for an OPCODE match. If found control goes to SERVICE. If an illegal opcode was fetched, then it goes to ERROR.

SERVICE indexes the CYCLE table to get specific information about the current opcode and then jumps to your custom test/evaluate/count routine. The OPCODE has been found. In this example, the routine ACCYC is used to count the cycles in the instruction being executed.

PROCESS is the return point from the custom service routine. It starts the actual processing of the current istruction. If, as indicated by a plus value in the A reg (from the CYCLE table), the instruction can be directly executed, then the program goes to TRANSF which completes the instruction execution.

SUB through **FORWRD** are the routines that service instructions that may not be directly executed. These are the instructions that modify the real program counter: JSR, JMP, JMP (X), RTS, RTI, and the conditional branches BEQ, BNE, BCC, BCS, BMI, BPL, BVC and BVS. Each of these instructions requires special processing to calculate the new program counter. This is handled by the various routines starting with SUB and ending with FORWRD. Once the new program counter has been calculated and set into PCTR then the instruction has effectively been executed! The program now goes back to FETCH for the next instruction at the new PC address. The call to subroutine OVER is specific to the TALLY cycle counting example and increments the count to reflect the extra cycle taken in crossing a page boundary. If your custom routine does not require special processing on page boundary crossings, then simply replace OVER with an RTS.

TRANSF to **EXBUF** are the real 'guts' of this program. This is where all of the instructions, except for the JMPs and BRANCHes handled above, are processed. The CYCLES table contains important information about each instruction. This is in the form:

Bit	Use in Cycles
01	Number of cycles
02	used by the
04	instruction
08	Number of operand
10	bytes in instruction
20	If set add X reg to indexed address, else add Y reg
40	If set check if page boundary crossed
80	If set do not directly execute the opcode

TRANSF moves the complete instruction to the three byte EXBUF and

FILLED pads with NOP's if the instruction is less than three bytes long.

POINT calculates the address of the next instruction.

OVERPG checks for indexed instructions and branches to RUN if not indexed.

SCAN2 checks for mode and branches to IND for indirect indexed mode.

ADDY,ADDX service a simple index instruction by modifying the address in the EXBUF and then go to RUN.

IND fixes up the address for the indirect indexed mode.

RUN restores all of the registers that were saved on entry.

EXBUF now contains the correct instruction to execute. It is executed and then drops through to the next code which saves all of the registers and then goes back to FETCH the next instruction.

BREAK is a BRK command that is executed when CONTROL encounters an RTI instruction. This stops CONTROL and returns you to your microcomputer monitor.

Tables

TABIN contains index values into the main opcode table. This considerably speeds up the search for the correct opcode during execution.

INST1 contains the values of all valid opcodes.

CYCLES contains the significant information about each opcode as described in the table above.

INST2 contains the value of all opcodes that require special service on page boundary crossovers.

ADRMOD indicates the addressing mode for each of the opcodes in INST2. A $00 byte indicates Indirect Indexed; an $FF indicates Absolute Indexed.

Utility Specific Routines

ACCYC is the basic cycle counter mechanism used in the cycle counting utility. It simply adds the number of cycles for the current instruction to the TALLY counter, an eight byte counter. This is called by SERVICE. For your own utility, write code to service your requirements (disassembler, trace mode, or whatever) and have SERVICE jump to it. Return to the mainline program with a JMP PROCESS.

OVER is an additional cycle counter used for page boundary crossovers that add one cycle to the instruction. If your utility does not need extra service for page boundaries, simply replace OVER with an RTS.

ERROR is called when an invalid opcode is encountered or when the utility code detects an error. It can be as simple as a BRK to abort processing and return to the system monitor; it can sound a tone and then BRK; it can include an error correcting scheme; or whatever you desire.

Using CONTROL

A simple application of the **CONTROL** program is that of counting the number of cycles used by a machine level program or subroutine. If you program in assembly language there are times when you would like to know how many cycles your coding uses. This information is useful for comparing the efficiency of one algorithm to another and when writing interrupt service routines that have a limited amount of time to perform their operations. Using CONTROL with the two *cycle counting* support routines provided will compute the exact number of cycles, including page boundary crossover cycles, used by your program. CONTROL runs your coding, but slower since there are instructions executed between each of the instructions in your program.

To use the cycle counter you should have a debugging monitor to display and change memory locations. Load CONTROL with the support routines **ACCYC** and **OVER**. Load the program you wish to test. Put the starting address of your program in **PCTR** ($B1 in our assembled version, may be different in your customized version). The least significant byte (LSB) goes in PCTR, the most significant byte (MSB) in PCTR + 1. If you need to initialize the 6502 registers, do so by putting the values in the storage locations **ACC**, **XREG** and **YREG** (A, X and Y registers), **STREG** (status register) and **STKPTR** (stack pointer). These locations are at $0BF8 to $0BFC in our version. Note that when using this program to count the cycles used by an interrupt service routine, the operation of the service routine is by CONTROL and begun by you, not by an interrupt. After an RTI instruction is processed, the BRK at BREAK will be processed and CONTROL will stop.

Now you can run CONTROL in the cycle counting mode. When it stops, display TALLY, the eight bytes starting at $0BF0 in our version, to see how many cycles your program used. The LSB is in the highest address, $0BF7. If an illegal opcode was fetched or there isn't enough room in TALLY to accumulate the cycles, then the error handler at ERROR will cause a BRK. See the separate examples for having ERROR sound a tone on the Atari, Apple and Commodore 64.

Adapting CONTROL

The original version of CONTROL was written on an Atari. The version listed here was run on the Apple II. The only change required was the memory location of the program itself. For the Atari, change the origin to $0600 or any other available 1K RAM. The Page Zero equates are okay. For the Commodore 64, $C000 is a handy origin for the program. Since Page Zero on the C64 is pretty full, the locations that you choose may depend on what else you are running. If you are **not** using the cassette tape and RS-232 port, for example, then the current equates of $B1 to $B4 should be okay.

The best way to make the adaptation is to key source into your assembler, change the equates and origin and re-assemble. This will give you a working version of CONTROL that you can then easily modify for other services: trace, single-step, disassemble, trap and so forth. If you do not have an assembler, the listed code can be directly keyed in. Make sure that you change the instructions that have direct references (generally the instructions with a value of 08 to 0B in the third byte of the instruction), plus the high byte address of BREAK that is referenced in the very first instruction.

[Editor's Note: This "cycle counting" demonstration of the CONTROL program is only one very limited use of this powerful technique. If you find CONTROL useful and extend its operation, MICRO is eager to help you share your work with the rest of the world. We are reserving space for CONTROL enhancements and guarantee extremely rapid publication.]

MICRO

Listing 1

```
                      ; Set BASE for 1K program area
0800                   BASE     EQU $0800
0800                            ORG BASE
                      ; Equates. PCTR, TEMPLO and
                      ; TEMPHI must be on Page ZERO
00B1                   PCTR     EQU $B1
00B3                   TEMPLO   EQU PCTR+2
00B4                   TEMPHI   EQU PCTR+3
                      ; Other equates can be anywhere,
                      ; including Page ZERO
0BF0                   TALLY    EQU BASE+$3F0
0BF8                   ACC      EQU TALLY+8
0BF9                   XREG     EQU TALLY+9
0BFA                   YREG     EQU TALLY+$A
0BFB                   STREG    EQU TALLY+$B
0BFC                   STKPTR   EQU TALLY+$C
                      ; Entry and Initialization
                      ; Put BREAK address on stack for
                      ; call via RTI in user code
0800 A9 0A             START    LDA #BREAK/256
0802 AE FC 0B                   LDX STKPTR
0805 9A                         TXS
0806 48                         PHA
0807 A9 32                      LDA #BREAK&$00FF
0809 48                         PHA
080A A9 00                      LDA #0
080C 48                         PHA
080D BA                         TSX
080E 8E FC 0B                   STX STKPTR
                      ; Clear TALLY counter
0811 A2 07                      LDX #7
0813 A9 00                      LDA #0
0815 9D F0 0B          CLR      STA TALLY,X
0818 CA                         DEX
0819 10 FA                      BPL CLR
                      ; Main Loop. Get
                      ; OPCODE of current instruction
                      ; Lookup in tables
081B A0 00             FETCH    LDY #0
081D B1 B1                      LDA (PCTR),Y
081F 48                         PHA
0820 29 F0                      AND #$F0
0822 F0 05                      BEQ INDEX
0824 4A                         LSR A
0825 4A                         LSR A
0826 4A                         LSR A
0827 4A                         LSR A
0828 A8                         TAY
0829 BE 33 0A          INDEX    LDX TABIN,Y
082C 68                         PLA
082D E8                SCAN1    INX
082E DD 43 0A                   CMP INST1,X
0831 F0 05                      BEQ SERVICE
0833 B0 F8                      BCS SCAN1
0835 4C CA 0B                   JMP ERROR
                      ; Start service when OPCODE found
                      ; Save info from CYCLES table on
                      ; stack. The JMP ACCYC is for the
                      ; TALLY Counter. Other operations
                      ; could be used instead.
0838 BD DA 0A          SERVICE  LDA CYCLES,X
083B 48                         PHA
083C 4C 9F 0B                   JMP ACCYC
                      ; Now Process instruction. First
                      ; test if the instruction can be
                      ; directly executed. If so, go
                      ; to TRANSF to execute.
083F A0 00             PROCESS  LDY #0
0841 68                         PLA
0842 30 03                      BMI SUB
0844 4C 8A 09                   JMP TRANSF
                      ; Instructions that change the PC
                      ; counter must be individually
                      ; serviced.
                      ; Test for JSR = $20
0847 B1 B1             SUB      LDA (PCTR),Y
0849 C9 20                      CMP #$20
084B D0 33                      BNE ABSJMP
084D A5 B1                      LDA PCTR
084F 85 B3                      STA TEMPLO
```

Listing 1 *(continued)*

```
0851 A5 B2            LDA PCTR+1
0853 8D B4 00         STA TEMPHI
0856 C8               INY
0857 B1 B1            LDA (PCTR),Y
0859 48               PHA
085A C8               INY
085B B1 B1            LDA (PCTR),Y
085D 85 B2            STA PCTR+1
085F 68               PLA
0860 85 B1            STA PCTR
0862 A5 B3            LDA TEMPLO
0864 18               CLC
0865 69 02            ADC #2
0867 85 B3            STA TEMPLO
0869 90 03            BCC STACK
086B EE B4 00         INC TEMPHI
086E AD B4 00  STACK  LDA TEMPHI
0871 AE FC 0B         LDX STKPTR
0874 9A               TXS
0875 48               PHA
0876 A5 B3            LDA TEMPLO
0878 48               PHA
0879 BA               TSX
087A 8E FC 0B         STX STKPTR
087D 4C 1B 08         JMP FETCH
               ; Test JMP = $4C
0880 C9 4C     ABSJMP CMP #$4C
0882 D0 0F            BNE INDJMP
0884 C8               INY
0885 B1 B1            LDA (PCTR),Y
0887 48               PHA
0888 C8               INY
0889 B1 B1            LDA (PCTR),Y
088B 85 B2            STA PCTR+1
088D 68               PLA
088E 85 B1            STA PCTR
0890 4C 1B 08         JMP FETCH
               ; Test JMPI = $6C
0893 C9 6C     INDJMP CMP #$6C
0895 D0 30            BNE RTSUB
0897 C8               INY
0898 B1 B1            LDA (PCTR),Y
089A 48               PHA
089B C8               INY
089C B1 B1            LDA (PCTR),Y
089E 85 B2            STA PCTR+1
08A0 68               PLA
08A1 85 B1            STA PCTR
08A3 A0 00            LDY #0
08A5 B1 B1            LDA (PCTR),Y
08A7 48               PHA
08A8 A5 B1            LDA PCTR
08AA C9 FF            CMP #$FF
08AC F0 0B            BEQ RESET
08AE C8               INY
08AF B1 B1            LDA (PCTR),Y
08B1 85 B2            STA PCTR+1
08B3 68               PLA
08B4 85 B1            STA PCTR
08B6 4C 1B 08         JMP FETCH
08B9 A9 00     RESET  LDA #0
08BB 85 B1            STA PCTR
08BD B1 B1            LDA (PCTR),Y
08BF 85 B2            STA PCTR+1
08C1 68               PLA
08C2 85 B1            STA PCTR
08C4 4C 1B 08         JMP FETCH
               ; Test RTS = $60
08C7 C9 60     RTSUB  CMP #$60
08C9 D0 17            BNE RTINT
08CB AE FC 0B         LDX STKPTR
08CE 9A               TXS
08CF 68               PLA
08D0 85 B1            STA PCTR
08D2 68               PLA
08D3 85 B2            STA PCTR+1
08D5 E6 B1            INC PCTR
08D7 D0 02            BNE CNGPTR
08D9 E6 B2            INC PCTR+1
08DB BA        CNGPTR TSX
08DC 8E FC 0B         STX STKPTR
08DF 4C 1B 08         JMP FETCH
               ; Test RTI = $40
08E2 C9 40     RTINT  CMP #$40
08E4 D0 15            BNE BRANCH
08E6 AE FC 0B         LDX STKPTR
08E9 9A               TXS
08EA 68               PLA
08EB 8D FB 0B         STA STREG
08EE 68               PLA
08EF 85 B1            STA PCTR
08F1 68               PLA
08F2 85 B2            STA PCTR+1
08F4 BA               TSX
08F5 8E FC 0B         STX STKPTR
08F8 4C 1B 08         JMP FETCH
               ; Must be a conditional Branch
08FB AD FB 0B  BRANCH LDA STREG
08FE 48               PHA
08FF B1 B1            LDA (PCTR),Y
               ; Test BEQ = $F0
0901 C9 F0            CMP #$F0
0903 D0 05            BNE BR1
0905 28               PLP
0906 F0 49            BEQ TRUE
0908 D0 39            BNE FALSE
               ; Test BNE = $D0
090A C9 D0     BR1    CMP #$D0
090C D0 05            BNE BR2
090E 28               PLP
090F D0 40            BNE TRUE
0911 F0 30            BEQ FALSE
               ; Test BCC = $90
0913 C9 90     BR2    CMP #$90
0915 D0 05            BNE BR3
0917 28               PLP
0918 90 37            BCC TRUE
091A B0 27            BCS FALSE
               ; Test BCS = $B0
091C C9 B0     BR3    CMP #$B0
091E D0 05            BNE BR4
0920 28               PLP
0921 B0 2E            BCS TRUE
0923 90 1E            BCC FALSE
               ; Test BMI = $30
0925 C9 30     BR4    CMP #$30
0927 D0 05            BNE BR5
0929 28               PLP
092A 30 25            BMI TRUE
092C 10 15            BPL FALSE
               ; Test BPL = $10
092E C9 10     BR5    CMP #$10
0930 D0 05            BNE BR6
0932 28               PLP
0933 10 1C            BPL TRUE
0935 30 0C            BMI FALSE
               ; Test BVC = $50
0937 C9 50     BR6    CMP #$50
0939 D0 05            BNE BR7
093B 28               PLP
```

Listing 1 *(continued)*

```
Ø93C 5Ø 13              BVC TRUE
Ø93E 7Ø Ø3              BVS FALSE
                ; Must be BVS = $7Ø
Ø94Ø 28          BR7    PLP
Ø941 7Ø ØE              BVS TRUE
                ; On branch condition FALSE, simply
                ; set PC counter to next instruction
Ø943 A5 B1       FALSE  LDA PCTR
Ø945 18                 CLC
Ø946 69 Ø2              ADC #2
Ø948 85 B1              STA PCTR
Ø94A 9Ø 2C              BCC FCH
Ø94C E6 B2              INC PCTR+1
Ø94E 4C 1B Ø8           JMP FETCH
                ; On branch condition TRUE, calculate
                ; new PC relative to current address
Ø951 C8          TRUE   INY
Ø952 B1 B1              LDA (PCTR),Y
Ø954 48                 PHA
Ø955 A5 B1              LDA PCTR
Ø957 18                 CLC
Ø958 69 Ø2              ADC #2
Ø95A 85 B1              STA PCTR
Ø95C 9Ø Ø2              BCC DIRECT
Ø95E E6 B2              INC PCTR+1
                ; Test branch direction
Ø96Ø 68          DIRECT PLA
Ø961 1Ø 18              BPL FORWRD
                ; Backward branch service
Ø963 49 FF              EOR #$FF
Ø965 18                 CLC
Ø966 69 Ø1              ADC #1
Ø968 85 B3              STA TEMPLO
Ø96A A5 B1              LDA PCTR
Ø96C 38                 SEC
Ø96D E5 B3              SBC TEMPLO
Ø96F 85 B1              STA PCTR
Ø971 BØ Ø5              BCS FCH
Ø973 C6 B2              DEC PCTR+1
Ø975 2Ø BA ØB           JSR OVER
Ø978 4C 1B Ø8    FCH    JMP FETCH
                ; Forward branch service
Ø97B 18          FORWRD CLC
Ø97C 65 B1              ADC PCTR
Ø97E 85 B1              STA PCTR
Ø98Ø 9Ø F6              BCC FCH
Ø982 E6 B2              INC PCTR+1
Ø984 2Ø BA ØB           JSR OVER
Ø987 4C 1B Ø8           JMP FETCH
                ; Move current instruction to
                ; buffer for execution. Use
                ; Opcode information from table
                ; to determine number of bytes of
                ; instruction to move to buffer
Ø98A 85 B3       TRANSF STA TEMPLO
Ø98C B1 B1              LDA (PCTR),Y
Ø98E 8D 1A ØA           STA EXBUF
Ø991 C8                 INY
Ø992 A5 B3              LDA TEMPLO
Ø994 29 18              AND #$18
Ø996 4A                 LSR A
Ø997 4A                 LSR A
Ø998 4A                 LSR A
Ø999 48                 PHA
Ø99A AA                 TAX
Ø99B CA          MOVE   DEX
Ø99C 3Ø Ø9              BMI FILLED
Ø99E B1 B1              LDA (PCTR),Y
Ø9AØ 99 1A ØA           STA EXBUF,Y
Ø9A3 C8                 INY
Ø9A4 4C 9B Ø9           JMP MOVE
                ; If less than three bytes of
                ; instruction, pad with NOP's
Ø9A7 CØ Ø3       FILLED CPY #3
Ø9A9 FØ ØA              BEQ POINT
Ø9AB A9 EA              LDA #$EA
Ø9AD 99 1A ØA    PUT    STA EXBUF,Y
Ø9BØ C8                 INY
Ø9B1 CØ Ø3              CPY #3
Ø9B3 DØ F8              BNE PUT
                ; Calculate address of next
                ; instruction
Ø9B5 68          POINT  PLA
Ø9B6 18                 CLC
Ø9B7 69 Ø1              ADC #1
Ø9B9 65 B1              ADC PCTR
Ø9BB 85 B1              STA PCTR
Ø9BD 9Ø Ø2              BCC OVERPG
Ø9BF E6 B2              INC PCTR+1
                ; Use Opcode info from table to
                ; see if Page boundary check is
                ; necessary.
Ø9C1 A5 B3       OVERPG LDA TEMPLO
Ø9C3 29 4Ø              AND #$4Ø
Ø9C5 FØ 41              BEQ RUN
                ; Service indexed instructions
Ø9C7 AD 1A ØA           LDA EXBUF
Ø9CA E8          SCAN2  INX
Ø9CB DD 71 ØB           CMP INST2,X
Ø9CE FØ Ø2              BEQ MODE
Ø9DØ BØ F8              BCS SCAN2
Ø9D2 BD 88 ØB    MODE   LDA ADRMOD,X
Ø9D5 FØ 1B              BEQ IND
Ø9D7 A5 B3              LDA TEMPLO
Ø9D9 29 2Ø              AND #$2Ø
Ø9DB DØ Ø6              BNE ADDX
                ; Add Y reg to operand
Ø9DD AD FA ØB           LDA YREG
Ø9EØ 4C E6 Ø9           JMP ADDY
                ; Add X reg to operand
Ø9E3 AD F9 ØB    ADDX   LDA XREG
Ø9E6 18          ADDY   CLC
Ø9E7 6D 1B ØA           ADC EXBUF+1
Ø9EA 9Ø 1C              BCC RUN
Ø9EC 2Ø BA ØB           JSR OVER
Ø9EF 4C Ø8 ØA           JMP RUN
                ; Indirect Indexed Address mode
Ø9F2 AD 1B ØA    IND    LDA EXBUF+1
Ø9F5 85 B3              STA TEMPLO
Ø9F7 A9 ØØ              LDA #Ø
Ø9F9 8D B4 ØØ           STA TEMPHI
Ø9FC A8                 TAY
Ø9FD B1 B3              LDA (TEMPLO),Y
Ø9FF 18                 CLC
ØAØØ 6D FA ØB           ADC YREG
ØAØ3 9Ø Ø3              BCC RUN
ØAØ5 2Ø BA ØB           JSR OVER
                ; Restore all registers
ØAØ8 AE FC ØB    RUN    LDX STKPTR
ØAØB 9A                 TXS
ØAØC AD FB ØB           LDA STREG
ØAØF 48                 PHA
ØA1Ø 28                 PLP
ØA11 AE F9 ØB           LDX XREG
ØA14 AC FA ØB           LDY YREG
ØA17 AD F8 ØB           LDA ACC
                ; Execute direct instruction
                ; stored in next three bytes
ØA1A ØØ ØØ ØØ    EXBUF  BYT Ø,Ø,Ø
                ; Save all registers
ØA1D 8D F8 ØB           STA ACC
ØA2Ø 8C FA ØB           STY YREG
```

Listing 1 *(continued)*

```
ØA23 8E F9 ØB          STX XREG
ØA26 Ø8                PHP
ØA27 68                PLA
ØA28 8D FB ØB          STA STREG
ØA2B BA                TSX
ØA2C 8E FC ØB          STX STKPTR
                ; And back to Main Loop
ØA2F 4C 1B Ø8          JMP FETCH
                ; Come here when an RTI is encountered
                ; in the User Program being tested
ØA32 ØØ         BREAK  BRK
                ; Index values to speed opcode search
ØA33 FF Ø8 1Ø   TABIN  BYT $FF,8,$1Ø,$1B
ØA37 23 2D 35          BYT $23,$2D,$35,$3F
ØA3B 47 5Ø 59          BYT $47,$5Ø,$59,$65
ØA3F 7Ø 7B 83          BYT $7Ø,$7B,$83,$8E
                ; All of the valid opcodes
ØA43 ØØ Ø1 Ø5   INST1  BYT Ø,1,5,6,8
ØA48 Ø9 ØA ØD          BYT 9,$ØA,$ØD,$ØE
ØA4C 1Ø 11 15          BYT $1Ø,$11,$15,$16
ØA5Ø 18 19 1D          BYT $18,$19,$1D,$1E
ØA54 2Ø 21 24          BYT $2Ø,$21,$24,$25
ØA58 26 28 29          BYT $26,$28,$29,$2A
ØA5C 2C 2D 2E          BYT $2C,$2D,$2E
ØA5F 3Ø 31 35          BYT $3Ø,$31,$35,$36
ØA63 38 39 3D          BYT $38,$39,$3D,$3E
ØA67 4Ø 41 45          BYT $4Ø,$41,$45,$46,$48
ØA6C 49 4A 4C          BYT $49,$4A,$4C,$4D,$4E
ØA71 5Ø 51 55          BYT $5Ø,$51,$55,$56
ØA75 58 59 5D          BYT $58,$59,$5D,$5E
ØA79 6Ø 61 65          BYT $6Ø,$61,$65,$66,$68
ØA7E 69 6A 6C          BYT $69,$6A,$6C,$6D,$6E
ØA83 7Ø 71 75          BYT $7Ø,$71,$75,$76
ØA87 78 79 7D          BYT $78,$79,$7D,$7E
ØA8B 81 84 85          BYT $81,$84,$85,$86
ØA8F 88 8A 8C          BYT $88,$8A,$8C,$8D,$8E
ØA94 9Ø 91 94          BYT $9Ø,$91,$94,$95
ØA98 96 98 99          BYT $96,$98,$99,$9A,$9D
ØA9D AØ A1 A2          BYT $AØ,$A1,$A2,$A4
ØAA1 A5 A6 A8          BYT $A5,$A6,$A8,$A9
ØAA5 AA AC AD          BYT $AA,$AC,$AD,$AE
ØAA9 BØ B1 B4          BYT $BØ,$B1,$B4,$B5
ØAAD B6 B8 B9          BYT $B6,$B8,$B9,$BA
ØAB1 BC BD BE          BYT $BC,$BD,$BE
ØAB4 CØ C1 C4          BYT $CØ,$C1,$C4,$C5
ØAB8 C6 C8 C9          BYT $C6,$C8,$C9,$CA
ØABC CC CD CE          BYT $CC,$CD,$CE
ØABF DØ D1 D5          BYT $DØ,$D1,$D5,$D6
ØAC3 D8 D9 DD          BYT $D8,$D9,$DD,$DE
ØAC7 EØ E1 E4          BYT $EØ,$E1,$E4,$E5
ØACB E6 E8 E9          BYT $E6,$E8,$E9,$EA
ØACF EC ED EE          BYT $EC,$ED,$EE
ØAD2 FØ F1 F5          BYT $FØ,$F1,$F5,$F6
ØAD6 F8 F9 FD          BYT $F8,$F9,$FD,$FE
                ; Opcode information bytes
ØADA Ø7 ØE ØB   CYCLES BYT 7,$ØE,$ØB,$ØD,3
ØADF ØA Ø2 14          BYT $ØA,2,$14,$16
ØAE3 8B 4D ØC          BYT $8B,$4D,$ØC,$ØE
ØAE7 Ø2 54 74          BYT 2,$54,$74,$17
ØAEB 96 ØE ØB          BYT $96,$ØE,$ØB,$ØB,$ØD,4
ØAF1 ØA Ø2 14          BYT $ØA,2,$14,$14,$16
ØAF6 8B 4D ØC          BYT $8B,$4D,$ØC,$ØE
ØAFA Ø2 54 74          BYT 2,$54,$74,$17
ØAFE 86 ØE ØB          BYT $86,$ØE,$ØB,$ØD,3
ØBØ3 ØA Ø2 93          BYT $ØA,2,$93,$14,$16
ØBØ8 8B 4D ØC          BYT $8B,$4D,$ØC,$ØE
ØBØC Ø2 54 74          BYT 2,$54,$74,$17
ØB1Ø 86 ØE ØB          BYT $86,$ØE,$ØB,$ØD,4
ØB15 ØA Ø2 95          BYT $ØA,2,$95,$14,$16
ØB1A 8B 4D ØC          BYT $8B,$4D,$ØC,$ØE
ØB1E Ø2 54 74          BYT 2,$54,$74,$17
ØB22 ØE ØB ØB          BYT $ØE,$ØB,$ØB,$ØB
ØB26 Ø2 Ø2 14          BYT 2,2,$14,$14,$14
ØB2B 8B ØE ØC          BYT $8B,$ØE,$ØC,$ØC
ØB2F ØC Ø2 15          BYT $ØC,2,$15,2,$15
ØB34 ØA ØE ØA          BYT $ØA,$ØE,$ØA,$ØB
ØB38 ØB ØB Ø2          BYT $ØB,$ØB,2,$ØA
ØB3C Ø2 14 14          BYT 2,$14,$14,$14
ØB4Ø 8B 4D ØC          BYT $8B,$4D,$ØC,$ØC
ØB44 ØC Ø2 54          BYT $ØC,2,$54,2,$74
ØB49 74 54 ØA          BYT $74,$54,$ØA,$ØE
ØB4D ØB ØB ØD          BYT $ØB,$ØB,$ØD,2,$ØA
ØB52 Ø2 14 14          BYT 2,$14,$14,$16
ØB56 8B 4D ØC          BYT $8B,$4D,$ØC,$ØE
ØB5A Ø2 54 74          BYT 2,$54,$74,$17
ØB5E ØA ØE ØB          BYT $ØA,$ØE,$ØB,$ØB
ØB62 ØD Ø2 ØA          BYT $ØD,2,$ØA,2
ØB66 14 14 16          BYT $14,$14,$16
ØB69 8B 4D ØC          BYT $8B,$4D,$ØC,$ØE
ØB6D Ø2 54 74          BYT 2,$54,$74,$17
                ; Opcodes that require page boundary
                ; crossover check
ØB71 11 19 1D   INST2  BYT $11,$19,$1D,$31
ØB75 39 3D 51          BYT $39,$3D,$51,$59,$5D
ØB7A 71 79 7D          BYT $71,$79,$7D,$B1
ØB7E B9 BC BD          BYT $B9,$BC,$BD,$BE
ØB82 D1 D9 DD          BYT $D1,$D9,$DD
ØB85 F1 F9 FD          BYT $F1,$F9,$FD
                ; Addressing mode table
                ; Ø = Indirect Indexed addressing
                ; FF = Absolute Indexed addressing
ØB88 FF ØØ ØØ   ADRMOD BYT $FF,Ø,Ø,$FF
ØB8C ØØ ØØ FF          BYT Ø,Ø,$FF,Ø,Ø
ØB91 FF ØØ ØØ          BYT $FF,Ø,Ø,$FF
ØB95 ØØ ØØ ØØ          BYT Ø,Ø,Ø,Ø
ØB99 FF ØØ ØØ          BYT $FF,Ø,Ø,$FF,Ø,Ø
                ; Service specific code goes here
                ; This version is a cycle counter
ØB9F A2 Ø7      ACCYC  LDX #7
ØBA1 29 Ø7             AND #7
ØBA3 18                CLC
ØBA4 7D FØ ØB          ADC TALLY,X
ØBA7 9D FØ ØB          STA TALLY,X
ØBAA 9Ø ØB             BCC ACCEND
ØBAC CA         ADDU   DEX
ØBAD 1Ø Ø3             BPL PROC
ØBAF 4C CA ØB          JMP ERROR
ØBB2 FE FØ ØB   PROC   INC TALLY,X
ØBB5 FØ F5             BEQ ADDU
ØBB7 4C 3F Ø8   ACCEND JMP PROCESS
                ; Specific service to count cycles
                ; This increments the count when a
                ; page boundary is crossed.
                ; Most routines could just put a
                ; RTS in place of OVER
ØBBA A2 Ø7      OVER   LDX #7
ØBBC FE FØ ØB   TAL    INC TALLY,X
ØBBF DØ Ø8             BNE LEAVE
ØBC1 CA                DEX
ØBC2 1Ø F8             BPL TAL
ØBC4 68                PLA
ØBC5 68                PLA
ØBC6 4C CA ØB          JMP ERROR
ØBC9 6Ø         LEAVE  RTS
                ; Machine and Routine specific error
                ; handler. Can be just BRK.
ØBCA ØØ         ERROR  BRK
                ; That's all !
ØBCB                   END
```

```
                ; Commodore 64 Beeper
                ;
                ; Note: There is not enough
                ; room at the end of the main
                ; program for this code since
                ; ØBFØ ... is used for storage.
                ; Put a JMP ERRORX at the current
                ; ERROR BRK location pointing to
                ; this code which may be relocated
                ; in any available memory.

ØØØØ A9 ØØ      ERRORX  LDA #$ØØ
ØØØ2 AA                 TAX
ØØØ3 9D ØØ D4   CLEARS  STA $D4ØØ,X
ØØØ6 E8                 INX
ØØØ7 EØ 19              CPX #$19
ØØØ9 DØ F8              BNE CLEARS
ØØØB A9 Ø9              LDA #$Ø9
ØØØD 8D Ø5 D4           STA $D4Ø5
ØØ1Ø A9 ØF              LDA #$ØF
ØØ12 8D 18 D4           STA $D418
ØØ15 A9 B1              LDA #$B1
ØØ17 8D ØØ D4           STA $D4ØØ
ØØ1A A9 19              LDA #$19
ØØ1C 8D Ø1 D4           STA $D4Ø1
ØØ1F A9 21              LDA #$21
ØØ21 8D Ø4 D4           STA $D4Ø4
ØØ24 A2 ØØ              LDX #$ØØ
ØØ26 AØ ØØ              LDY #$ØØ
ØØ28 C8         WAITS   INY
ØØ29 DØ FD              BNE WAITS
ØØ2B E8                 INX
ØØ2C DØ FA              BNE WAITS
ØØ2E ØØ                 BRK



                ; Atari Version of Beeper

ØBCA A9 ØØ       ERROR  LDA #Ø
ØBCC 8D Ø8 D2           STA $D2Ø8
ØBCF A9 Ø3              LDA #3
ØBD1 8D ØF D2           STA $D2ØF
ØBD4 A9 A8              LDA #$A8
ØBD6 8D Ø1 D2           STA $D2Ø1
ØBD9 A9 79              LDA #121
ØBDB 8D ØØ D2           STA $D2ØØ
ØBDE ØØ                 BRK



                ; Apple II Version of Beeper

ØBCA 2Ø E4 FB    ERROR  JSR $FBE4
ØBCD ØØ                 BRK
;
```

feature

SIXTEEN BIT 68000 SUPERMICROS

by Paul Lamar and Richard Finder

Editor's Note: While we normally do not publish articles that are essentially "one man's opinion", we are making an exception in this case because 1) it touches on a very important area, the 68000, and 2) they are eminantly qualified to talk about the issues.

Two seasoned computerists share their insights into the world of the 16 bit 68000 Supermicro.

It may have been a result of reading an over-abundance of IBM PC ads that caused people, without knowledge of assembly language or microprocessor architecture, to blatantly predict that MS-DOS on the eight bit 8088 chip will become the measure by which all sixteen bit microcomputers will be judged during the coming decade. That view is simply wrong, and such comments (especially by people who should know better) may be the result of an understandable impatience with the performance of slow, memory limited, eight bit microcomputers --but to declare that the 80XXX is going to be the de facto industry standard is short-sighted at best, and misleading at worst.

For those preparing to buy a serious microcomputer for the first time (not just an elaborate toy), be aware that even though the IBM PC and all its clones use 8088 chips, they use them as eight bit CPUs. (IBM claims the 8088 in the PC is sixteen bits, but it just isn't so. The 1983 Intel Microprocessor and Peripheral Handbook clearly states, on page 3-79, that the 8088 is an eight bit microprocessor, and they should know. They invented the chip). IBM justifies this claim by citing the 16 bit internal registers in the 8088. The Commodore 6502 used in the Apple and the Commodore 64 has one, sixteen bit register (the program counter). The 6502 is not called a sixteen bit microprocessor. The Motorola 6809 used in the Radio Shack Color Computer has six, sixteen bit registers and it is not called a 16 bit microprocessor. Why call an 8088 a sixteen bit microprocessor?

Watch Large Computer Corps. (LCCs) carefully; they take advantage of ignorance every chance they get. Rather than try to educate the user, the LCC uses seduction to persuade the buyer into a purchase not suited to the individual's needs or desires. A corollary of the business maxim "buy low, sell high" is "sell as little as possible for as much as you can get". It is the buyer's responsibility (in computers, as well as cars, houses, and health insurance) to learn something about microcomputers before writing out that first check. Any LCC ad which doesn't set forth facts about number-of-characters-on-the-screen, disk storage capacity, RAM, ROM, megabytes and megahertz is hiding something (probably mediocre performance or operational deficiencies).

There is a common myth that speed and power in a microcomputer are not really necessary when "all you are going to use that microcomputer for is word processing". A fast typist types about 60 words a minute. If each word is an average of five characters in length; that means that one character is going into the computer every 200,000 micro seconds. When you are typing characters into a wordprocessing program, it takes a typical microprocessor and program about 10,000 micro seconds to process that character. The other 190,000 micro seconds the processor is twiddling its thumbs so to speak. Why not put that time to good use by a fast and powerful microprocessor. How many characters of spelling or grammar could that micro check in those remaining 190,000 micro seconds?

It's not a matter of being a microcomputer speed freak, but of not wanting to waste time while some infernal machine which knows nothing about time and couldn't care less does something useful. "Disgruntled" is an eleven-letter word for the owner of a micro-word processor who has to look up a word in a dictionary because the human works faster than the computer. There is nothing more useless than a $150 spelling checker which isn't used -- because the machine is too slow.

The dream word processing program is one which checks the spelling of the word as it's being typed in. Ideally, it could not only check the spelling of the word, but could finish writing out the word. For example, the writer would begin the word "spelli"; the computer would fill in "ng" and the cursor would jump to the next word position (there's only one word spelled "spelling"). Of course, turning off such a feature would be a necessary option. As an alternative, a misspelling could cause a word to be flagged or prompt a beep, and optionally show the suggested correction as part of a dictionary in a window, along with the definition(s) of the word.

Computers are tools to increase productivity. An automobile manufacturer who designed a factory to produce automobiles 20% slower than that of the competition faces business failure. As a writer, accountant, or business manager, why buy an eight bit 8088 based microcomputer that is one fourth as fast as a true sixteen bit 68000 based supermicro?

As was said by the philosopher, Dionysius of Halicarnassus, "...history is philosophy learned from examples". The philosophical point espoused here is the superiority of the 68000 chip for state-of-the-art microprocessing. My own history (which brought me to this point of view), is that several years ago I was part owner of one of the first Apple peripheral and software manufacturing firms. Our company bought one of the first two hundred Apple II processor boards made, which was delivered with 4K of RAM, with no keyboard, power supply or case. Documentation consisted of a printed color brochure and some photocopied pages in a plastic-covered binder off a drug store shelf. No system monitor source listing came with the computer; a complaint to Steve Wozniak brought a photocopy of it.

Before the Apple II, I wire wrapped an Intel 4040 and RCA CMOS 1801 (not an 1802) microprocessors. The 4040 was a nightmare with many different silicon technologies that required voltage level shifting among the various required chips. Intel's promotion literature did not mention this. Only after you bought the $100 chip and received the data sheet did this become apparent. I bought an early 1K RAM, 2K ROM, MOS Technology KIM-1.

The KIM-1 was a revelation and very easy to use. I wrote a real-time, multi-tasking, interrupt-driven program on the KIM-1 using the KIM-1's hex keypad and 6 digit LEDs. That program required six months to write, yet it was only 2K bytes in length (I kludged on another 1K RAM). I designed and manufactured an industrial microcomputer called the SUPERKIM which we are still manufacturing.

We bought the Apple II board because we needed a more powerful microcomputer than the KIM-1 to write 6502 assembly language software. (Writing and assembling programs is one of the most demanding tasks you can ask of any computer.) We attached a homemade power supply, a surplus keyboard, and a used video monitor to our new Apple II board -- and it worked. We wrote a crude printer driver routine using the built-in miniassembler for a South West Technical Products PR 40 printer, then designed a very simple printer interface board for the Apple II. To my knowledge, this was the first printer interface ever sold for the Apple II.

We searched for a symbolic assembler to use on the Apple. (We where not sure at that time what a symbolic assembler was, but our friends assured us that it was something we needed.) A symbolic assembler allows you to jump to a name (symbol) of a routine within a program, rather than to its address. (Jumping to the address of a routine is what you do in BASIC when you say GOTO 1010.) Unlike an address, the name of a routine doesn't change regardless of how much code you put in front of it. Most symbolic assemblers automatically calculate the branch addresses as well, unlike the mini assembler in ROM, in the Apple II.

Bob Bishop and I typed in a four character symbolic assembler written by Carl Moser (lately of Eastern House Software), and Bob (later of Apple Vision fame) made it work. Our assembler was a big step above the Apple mini-assembler, and we sold many of those four-character symbolic assemblers. This too was a first for the Apple II.

So it went for several years until other computers arrived on the market and we slowly began to realize what we were missing: eighty columns on the display, a screen editor, larger disk and RAM storage and speed. Eighty columns was particularly missed when writing assembly language text files as there was no room for comments on the right side of the screen. We needed larger disk storage because a 2K assembly language program occupies about 32K of commented text file on a disk. We did not want to utilize any of the third party solutions to these problems due to potential incompatibility with our then present software--and there was a tendency of Apple II software vendors to copy-protect their product, making their software impossible to store on hard disk or make back up copies.

By this time we had become authorized dealers for Apple, Commodore, Zenith and Kaypro, in addition to manufacturing and selling our own CP/M, eight inch drive, Z80 system; all of these were too slow. The Commodore 8032/8050 was the best of the bunch thanks to the legendary Chuck Peddle, designer of the 6502 (then working for Commodore). It had an amazing 500K on each single sided five-inch disk. Poor Chuck made a big mistake on the Victor when he designed in the eight bit 8088 rather than the 68000 (he's now an ex-president of Victor--and Victor is in Chapter 11), apparently a victim of the IBM mystique. The Commodore 8032 lacked the speed or RAM memory desired to justify switching from the Apple II.

The imminent arrival of the Apple III carried hopes that it would have a 68000 microprocessor, but it had instead a 6502A microprocessor--only 143K on the disk, memory bank switching, and a steep price tag. Several computer store owners actually shouted epithets at Apple's Barry Zargoni when he introduced the Apple III at the pre-release dealer's meeting. Apple management ignored their dealers.

While the Apple III had a few hardware problems when it was first announced, those were not the main reason for its disappointing sales. In the very early days of the Apple II some wordprocessing programs--horribly slow-- were written with interpreted integer BASIC. An operating system, a high level language or a wordprocessing program written with a high level language (an HLL, such as BASIC) results in very slow performance. The only proper way is to use assembly language. Thus, the Apple III BASIC ran about the same speed as Applesoft on the Apple II despite the fact that the processor was twice as fast.

The Apple III BASIC was written with a HLL and compiled. There were no schematics or source listings provided for the Apple III, nor even instructions for using the built-in system monitor. How could we design peripherals or write assembly language software (or even fix it if it broke)? When the wonderful Apple II came out, it was accompanied by all these amenities. Furthermore, for the assembly language programmer, the Apple III's memory bank switching was a horrible feature. Memory bank switching stemmed from Apple's choice of the eight bit 6502A. Since the 6502A could directly address only 64K bytes, memory bank switching was necessary, and meant that the programmer had to keep track of which bank his subroutine was in (the one that he would like to call) and which bank he himself was in, when he called that subroutine to return to the bank in which he had been working. Such systems limit the practical size of a non-bank switched program to just 64K--but the Apple III had 256K of bank switched RAM!

Assume momentarily that a controlling operating system program is 16K bytes long. It can never be switched; that would be like jumping to an undefined area of memory with no meaningful program stored in it. Another 16K bytes is allocated to program modules which do different things, whether in the control system or elsewhere, and can be switched as needed. This leaves only 32K in a standard 64K system for text files. To search through a large dictionary, one must bank-switch that dictionary in from the disk or from another bank of RAM memory, 32K bytes at a time. The larger the program modules, the smaller the text files must be. Imagine the frustration of sorting something larger than 32K

Thus, the statement that memory bank switching was "horrible"; it's a piece of hardware designed to give an assembly language programmer nightmares, besides being slower than storage in a large linear address space, such as is available on the 68000. If only Apple had used the 68000 in the Apple III and had written the system software in assembly language they would now be in an unassailable position, instead of second place and dropping. (Significantly, they now use the 68000 in their MacIntosh, but have yet to introduce an operating system with any significant amount of software to match the chip... but that's a different story, having to do with the P-System).

In Motorola's sixteen bit 68000 microprocessor, the assembly language instructions set is similar to the 6502, but immensely more powerful. The 68000 is about one fourth as difficult to program in assembly language as the 6502, yet about four times faster to program for any given application. The 68000 was designed four or five years ago with thirty-two bit internal architecture, while Intel and Zilog were designing their sixteen bit microprocessors with sixteen bit internal architecture. Because the 68000 has thirty-two bit internal registers, including the address counter, it can address sixteen megabytes without memory bank switching.

A thirty-two bit address bus implies four gigabytes (four thousand megabytes) of address space, though only twenty-three address lines and upper and lower byte address strobes are brought outside the chip; hence sixteen megabytes. All of the following microprocessors can only address 64K without memory bank switching: eight bit Intel 8088, 80188; sixteen bit Zilog Z8000, or Intel 8086, 80186, 80286, 80386.

Intel advertises one megabyte-plus addressing on these last-mentioned chips because they built in that horrible bank switching circuitry. Intel calls it "segmenting", but the programmer still has to do the dirty work. The longest internal register these chips contain is sixteen bits, therefore, the most memory they can address is 64K bytes. For this and other reasons their assembly language instruction set is unorganized and inconsistent compared to the 68000. (Besides, the 68000 is twice as fast as a sixteen bit 8086--not to mention the much slower eight bit 8088 IBM uses in the IBM PC).

A fifty dollar 12.5 mhz 68000 is as fast as a $150,000 Digital Equipment Corporation (DEC) VAX 11/780 CPU. Furthermore, the VAX 11/780 can only address eight megabytes; the 68000 addresses sixteen megabytes. It may be hard to believe, but it's true. A sixteen megahertz version of the 68000 is in the sampling stages already.

Hardware floating point operations on the 68000 are three times faster than the 8086/8087 combination because National Semiconductor's 16081, high speed math chip (sixty-four bit floating point multiply in twenty three microseconds) works faster with the 68000 than with National's own sixteen bit microprocessor.* Software written for the present 68000 will have a long and useful life because it is upwardly compatible with the full 32 bit address (4 gigabytes) and data bus version of the 68000 (the Motorola 68020). Not only that, but the 68020 is four times faster than the 68000. Consequently, the 68020 has a three or four year head start on software compared to any other full 32 bit microprocessor. No other 32 bit microprocessor on the horizon is sufficiently better or faster than the 68020 to overcome the software lead the 68020 enjoys.

Unfortunately, greed is still around, and getting worse. Most large software houses think like this; "Knock it out with an HLL---nobody will notice how slow it is until after we make a killing". Such software houses therefore need increasingly faster microprocessors so they can justify writing new word-processing programs and operating systems in a new HLL, that was written in an old HLL.

About two years ago we read an ad in "Byte" for theSAGEsupermicro and contacted SAGE Computer for information. We were initially impressed because it came with the P-System, wordprocessing, spreadsheet, PASCAL and a 68000 macro assembler, along with an assortment of other software. When we saw the extensive documentation, the schematic, the memory map, the powerful system monitor in 16K byte EPROM, and the monitor source listing--in other words, a completely open system--we were sold.

The experience was like that of a few years before, when we were first introduced to the Apple II, except that with the SAGE we were given an extensive assortment of software and a built in printer interface just to start up our acquaintance. In short, we bought a SAGEand have been pleased with the supermicro to this day; it has proven its reliability and speed.

We use it with a 6502 macro cross assembler to write all our software for other uses, and for wordprocessing. We were even able to upload 6502 assembly language text files to the SAGE and cross assemble them after a few changes with the editor. (An unexpected bonus, most welcome). BASIC and PASCAL text files were also uploaded. The secret to doing this is to use the Apple II serial printer interface and a free utility on the P-system called "TEXTIN". The P-system, program editor's replace function is easily used to change 6502 assembly language pseudo-ops and Applesoft BASIC commands to conform to P-system language requirements.

Floppy disk access and load time (20K per sec) execute on the SAGE about ten times faster than on the Apple II disk operating system (DOS),

(continued on page 51)

---

* *DTACK GROUNDED, The Journal of Simple 68000 Systems. Issue 24, October-1983. DTACK GROUNDED 1415 E. McFadden, Ste. F, Santa Ana, CA 92705*

Programming with Macros

by Patricia Westerfield

You can make your assembly language more efficient, cleaner, easier to debug.

Key to Understanding

Macros are a group of commands in assembly language assigned a mnemonic which can then be used alone in a program. When the program is run, those commands assigned to the Macro mnemonic are processed in a manner similar to a subroutine. Macros become a tremendously powerful tool for the programmer when the way that they can be used is understood. Assembly language programming is often avoided because of its simplistic and tedious nature. But for many programmers it has become a necessity because of memory limitations and the requirement for fast programs. Maintaining a program or system written in any language can be difficult and time consuming. Problems encountered are compounded when the program is written in assembly language. Macro instructions change this by enabling the programmer to retain the efficency of assembly language while providing the capacity to emulate some features of higher level languages.

Macros also alleviate debugging and other problems by bringing about a standardization of code. Operations used repeatedly throughout the program are handled in the same manner, and are, therefore, easily identified. The code is much shorter with mnemonically named macros and considerably easier to read. This, combined with the basic comment structure all assemblers provide, puts structured programming within the reach of every assembly language programmer.

Introduction

The techniques and examples described in this article use the ORCA/M Macro Assembler for the Apple II, from Hayden Software Co. The ORCA assembler has its own specific macro language, explained fully in the manual, which allows the programmer to write macros tailored specifically to his needs. But, because the system supplies over 150 macros with complete subroutine library support, the typical assembly language programmer will probably never need to write a macro. For this reason, this article will focus on the ways in which macros can be used to enhance and simplify assembly language programming, and not the symbolics of the macro language.

Replace HEX Addresses

The first, and perhaps simplest, reason to use a macro is to replace an easily forgotten address. The Apple monitor contains 32 subroutines, documented by Apple, for use by the assembly language programmer. These subroutines range from generating a carriage return to drawing a horizontal line of low resolution graphics blocks. To access these routines, the correct memory location or 6502 registers are loaded, followed by a jump to subroutine instruction and the hexadecimal number which is the subroutine's starting address. The Apple monitor will then perform the desired functions and return to the instruction immediately following that from which it was called.

The COUT macro is used to illustrate this point; it prints out the character contained in the A register. Without a macro, the code to initiate this subroutine would look like:

```
.
.
LDA #'A'
JSR $FDED
.
.
```

In this example the A register is loaded with the character 'A'. This is followed by the jump to subroutine call, which goes to the memory location $FDED where the subroutine in the Apple monitor performs the necessary instructions to print out the 'A' character.

To circumvent the problem of having to remember the 32 hexadecimal addresses needed to access the monitor subroutines, a macro can be used to replace the address with a short name which describes the function of the subroutine. This name is more easily remembered, saving the programmer time and reducing the chance of error. When using a macro to call the character out subroutine, the LDA and JSR instructions are replaced by a single macro:

```
.
.
COUT #'A'
.
.
```

Replace Repetitive Code

Another use of macros is to replace repetitive bits of code that are too small to require writing a separate subroutine. Suppose a program required getting the characters from a line one at a time. The code needed to get the next character from a line of input and load it into the A register would need to be duplicated in several places throughout the program.

Below is an example of what the code to perform this function might look like:

```
        .
        .
        INC   CCHAR
        LDX   CCHAR
        LDA   LINE,X
        .
```

A line of input can contain up to 255 characters. In this example CCHAR (current character) is the index number of the position in the line the computer is looking at. The first instruction increments CCHAR so it is now pointing to the next character. Next, the line position of the character is loaded into the X register and then the character X is pointing to is loaded into the A register. A desirable alternative to writing these 3 lines of code numerous times in the program is to define a macro NCHR (next character) to perform this function. By using this macro each time a new character is needed, the number of lines of code the programmer will have to write, and later wade through when debugging, will be decreased significantly. The code to execute this would look like:

```
        .
        .
        NCHR
        .
        .
```

Define New Instructions

New instructions can also be written with macros to eliminate the requirement for many different instruction sequences to handle variations of an operation. The ADD macro is a case in point. Not only can variable parameters be passed, designating different numbers and locations to be operated on, but the macro will optimize the add by skipping unnecessary instructions.

The following code illustrates a typical two byte add in assembly language:

```
        .
        .
        CLC
        LDA   NUM1
        ADC   NUM2
        STA   NUM3
        LDA   NUM1+1
        ADC   NUM2+1
        STA   NUM3+1
        .
        .
```

The first step in performing the add operation is to clear the carry flag. In this example, the low bytes of the numbers contained in NUM1 and NUM2 are added together and stored in the location designated here as NUM3. This is followed by an add of the high bytes of the numbes contained in NUM1 and NUM2 which is stored in the high byte of NUM3. The total number of bytes needed to perform this add is 19 (assuming that no variables are in page zero).

The ORCA assembler provides an ADD macro which replaces these 7 lines of code with one while duplicating the above operation:

```
        ADD NUM1,NUM2,NUM3
```

The macro performs the same 2 byte add and stores the result in NUM3. The macro also required 19 bytes.

The ADD macro in ORCA will always do a 2 byte add, but when adding a 1 byte immediate number to a 2 byte number, the standard shortcut is automatically taken.

What follows is the code needed to add 4 to NUM1 without macros:

```
        .
        .
        CLC
        LDA   NUM1
        ADC   #4
        STA   NUM1
        BCC   PAST
        INC   NUM1+1
PAST    ANOP
        .
        .
```

After the carry flag has been cleared the 4 is added to the low byte of the number stored in the location NUM1. The next step is to increment the high byte of NUM1 if the first add resulted in an overflow. Notice that in this example the sum of the two numbers is returned to the location NUM1. The total number of bytes required to perform this operation is 16, assuming no zero page locations.

To illustrate the fact that the ADD macro will take the shortcut when applicable, the same ADD macro is used, this time with the GEN ON directive in place. This directive is provided with the ORCA assembler. When it is turned on at the beginning of the program all the lines generated by the macro expansion are printed in the output listing. These lines of code are preceded by a '+'. Notice that the following lines of code are basically the same as those above:

```
          .
          .
          ADD   NUM1.#4
+         CLC
+         LDA   NUM1
+         ADC   #<4
+         STA   NUM1
+         BCC   SL2
+         INC   NUM1+1
+SL2      ANOP
          .
          .
```

If the carry flag is clear after the low bytes of the two numbers are added together, the high byte of NUM1 is not incremented. Instead the assembler branches around this instruction to the label SL2 which is a ANOP (assembler no-operation). Because the ADD macro was able to recognize and use the standard assembly language shortcut, a savings of 3 bytes resulted. At first glance this may not appear to be a significant savings, but when the number of times these macros are used in a large program is taken into account, the savings in space and the speed up during assembly time become significant.

Notice that the two previous examples used the same ADD macro to perform two different types of add:

```
        ADD NUM1.NUM2.NUM3.
        ADD NUM1.#4
```

The ADD macro, like many other macros in ORCA, allows variable parameters to be passed to the macro. In the first example the result of the add is stored in NUM3. If a destination is not specified, as in the second case, the result is stored in the first location by default, in this case NUM1. This feature alone saves the programmer from having to code many different instruction sequences to do basically the same operation, thereby adding to the efficency of assembly language programming.

Shorten Code

Macros also shorten the number of lines of code in a program, making it easier to read and less prone to error. This also speeds up program development: an oft quoted result of several studies on programming is that a programmer programs a constant number of lines of code per hour, regardless of the language. By reducing the number of lines of code, program development speeds up.

The following statements load the address of a two byte number AD2 into AD1 least significant byte first:

```
        .
        .
        LDA   #< AD2
        STA   AD1
        LDA   #> AD2
        STA   AD1+1
        .
        .
```

These 4 lines of code can be replaced by the load address macro, LA:

```
        LA  AD1,AD2
```

thus performing the same function without extra lines of code.

Hide Confusing Code

A major advantage to programming in Pascal or another high level language, rather than in assembler, is the ability to give a function or procedure a name which clearly describes the operations being performed. Because of the simplistic nature of assembly language the purpose of even a few lines of code can become difficult to discern a very short time after the code has been written.

The ORCA PRINT macro hides what can be confusing lines of code while at the same time stating clearly the procedure to be performed. The macro is straightforward, emulating its BASIC counterpart by writing out the characters contained in ticmarks:

```
        PRINT 'A LINE OF OUTPUT'
```

would result in

```
        A LINE OF OUTPUT
```

printed out to the CRT or the printer, whichever was specified by the programmer. The expansion of this macro would look like:

```
            .
            .
            PRINT    'A LINE OF
                       OUTPUT'
+           JSR   SRITE
+           DC    H'8Ø',I1'L:SL2'
+SL2        DC    C'A LINE OF
                    OUTPUT'
            .
```

The macro statements generated, the ones preceded by the ' + ', show the steps the PRINT macro takes to perform its task. First a jump to subroutine call is made to SRITE, which is contained in the system library. This is followed by two DC (declare constant) assembler directives. These statements tell the subroutine the length of the output and whether or not a return needs to be generated after the line of output is printed. These three lines handle a number of tedious coding steps the programmer would be required to code if this macro was not available. The efficency of assembly language is retained, while at the same time it is possible to achieve some of the advantages of a higher level language.

Standardize Code

A great deal of confusion can be eliminated through standardization of code using Macros. Consider Fig. 1 and Fig. 2. Both of these subroutines perform the same task, that of printing a menu on the screen and accepting user inputs. Fig. 1 is written in straight assembly code, while Fig. 2 uses macros and and implements a simple commenting structure. An experienced assembly language programmer would be required to decipher the purpose of the code in Fig. 1. The macros and comments used in the example in Fig.2 enable the main points of the subroutine to be understood even by programmers unfamiliar with assembly language.

Alternate Instruction Sets

Another feature macros provide, useful to the advanced programmer, is the ability to write alternate instruction sets. An excellent example of this is a cross assembler which would allow code written using the ORCA 6502 Assembler to be run on another microprocessor such as the 6809. The gap between these instruction sets is bridged with macros.

The only major problem that arises when writing a cross assembler involves handling identical instructions which assemble differently on each microprocessor. To get a better idea of the problem, consider the RTS (return to subroutine) instruction on the 6502 and the 6809. The RTS on the 6502 is equivalent to a hex 60 while the RTS on the 6809 is the same as a hex 39. In order for the assembler to distinguish which RTS is meant to be used at a given time there must be a way to separate the instruction sets. The first way to solve this problem is to code all 6809 instructions in lowercase and leave all 6502 instructions in uppercase:

```
        rts
```

Another way is to precede each 6809 op code with an identifier, such as a '.':

```
        .RTS
```

Macro Libraries

The ORCA assember's macro library provides a collection of standard macros which can be used to perform common functions. Because these macros come with the system, they need need not be recoded for each program.

To use the macros effectively, the programmer builds a small library of the macros used in a particular program. This file takes very little time to colate and speeds up the assembly of the program. With a separate macro library the assembler only has to search through the macros needed by the program, and not the entire 150 macros provided with the system.

Subroutine Libraries

In order for macros to be of optimum use to the programmer they must be backed up with subroutine libraries. The reason for a subroutine library becomes apparent when the SUB (subtract) macro is compared to the MULT (multiply) macro. With the GEN ON directive in place at the beginning of the program the code the subtract macro would generate would look like:

```
     .
     .
     SUB   NUM1,NUM2
+    SEC
+    LDA   NUM1
+    SBC   NUM2
+    STA   NUM1
+    LDA   NUM1+1
+    SBC   NUM2+1
+    STA   NUM1+1
     .
     .
```

Compare this with the multiply macro:

```
     .
     .
     MULT  NUM1,NUM2
+    ANOP
+    LDA   NUM1      move NUM1 to
+    STA   M1L        mult reg
+    LDA   NUM1+1
+    STA   M1H
+    LDA   MUN2      move NUM2 to
+    STA   M3L        other mult reg
+    LDA   NUM2+1
+    STA   M3H
+    JSR   SMULT     perform
                     multiply
+    LDA   M1L       move answer to
+    STA   NUM1       NUM1
```

Listing 1

```
         KEEP    MENU,V1Ø
SMENU    START
PRBL     EQU     $F94A          PRINT BLANKS
HOME     EQU     $FC58          MONITOR
                                HOME ROUTINE
RDKEY    EQU     $FDOC          READ KEYBOARD
CROUT    EQU     $FD8E          DO CARRIAGE
                               RETURN
BELL     EQU     $FF3A          RING BELL

         JSR     SINIT
MN1      JSR     HOME           WRITE MENU
         LDX     2Ø-(MSG2-MSG1)/2
         JSR     PRBL
         LDX     MSG1
         LDA     MSG
         LDY     MSG2-MSG1
         JSR     SRITE
         JSR     CROUT
         JSR     CROUT
         LDX     MSG2
         LDA     MSG2
         LDY     MSG3-MSG2
         JSR     SRITE
         JSR     CROUT
         LDX     MSG3
         LDA     MSG3
         LDY     MSG4-MSG3
         JSR     SRITE
         JSR     CROUT
         LDX     MSG4
         LDA     MSG4
         LDY     MSG5-MSG4
         JSR     SRITE
         JSR     CROUT
         JSR     CROUT

GT1      JSR     RDKEY          GET A LEGAL
                                INPUT
         CMP     '1'
         BLT     GT2
         CMP     '4'
         BLT     GT3
GT2      JSR     BELL           ERROR FOUND
         JMP     GT1

GT3      CMP     '1'           CATALOG THE DISK
         BNE     GT4
         JSR     SCTLG
         JMP     MN1

GT4      CMP     '2'           LOAD A FILE
         BNE     GT5
         JSR     SLOAD
         JMP     MN1

GT5      RTS

MSG1     DC      C'MENU'
MSG2     DC      C'  1)          CATALOG'
MSG3     DC      C'  2)          LOAD A FILE'
MSG4     DC      C'  3)          QUIT'
MSG5     DS      0
         END

         KEEP    MENU,V1Ø
         MCOPY   MENU,MACROS,V1Ø
************************************************
*                                              *
*  SMENU - PRINT MENU ON SCREEN AND            *
*         ACCEPT USER INPUTS                   *
*                                              *
************************************************
*
SMENU    START
;
;        WRITE MENU
;
         JSR     SINIT
MN1      HOME
         PRINTC  'MENU'
         CROUT
         PRINT ' 1) CATALOG'
         CROUT
         PRINT ' 2) LOAD A FILE'
         CROUT

         PRINT ' 3) QUIT'
         CROUT
         CROUT
;
;  GET AND INTERPRET USER INPUT
;
GT1      RDKEY CRS  GET A LEGAL INPUT
         CMP     '1'
         BLT     GT2
         CMP     '4'
         BLT     GT3
GT2      BELL                   ERROR FOUND
         JMP     GT1

GT3      CMP     '1'           CATALOG THE DISK
         BNE     GT4
         JSR     SCTLG
         JMP     MN1

GT4      CMP     '2'           LOAD A FILE
         BNE     GT5
         JSR     SLOAD
         JMP     MN1

GT5      RTS                    QUIT
         END
```

```
+  LDA   M1H
+  STA   NUM1+1
      .
      .
```

Both of these macros generate these lines of code in the program at the place where they are use. Notice however that the SUB macro completed the entire operation in 7 lines of code. On the other hand the MULT macro merely set up the numbers given it in a standard format and called the multiply subroutine (JSR SMULT) to perform the calculation. The fact that a subroutine library was called from a macro means that the dozens of lines of code in that subroutine are not generated each time the macro is called; instead it is stored in one place in memory, thus cutting down the amount of memory the program requires in order to run.

The beauty of subroutine libraries lies in the fact that they are preassembled, and therefore never need to be assembled again. The subroutines required by the program are automatically linked in at assembly time by the link editor. For this reason no assembly time is lost on these subroutines.

The concept of subroutine libraries can be invaluable to a programming shop involved with software development. Only a central shop is required to develop and maintain the system libraries. Because not everyone has the source code to these libraries it becomes difficult to alter code, which is (presumably) known to be error free. At the same time interfacing with these routines becomes standard, thereby making the finished program easier to maintain and update.

Conclusions

The macros supplied with the ORCA system effectively extend the 6502 assembly language instruction set to over 200 instructions. Many instructions not commonly found in the 6502 assembly language, like PRINT and HOME, are now available to the programmer via macros. Areas where the processor was inadequate, including I/O and arithmetic, are handled with ease. Because of these added instructions, assembly language is no longer tedious and difficult to use. Instead, it approaches the simplicity of a higher level language.

Through macros, the full potential of programming in assembly language is reached. Macros enable the programmer to write fewer lines of code to accomplish a given task, and to do so in a precise and straightforward way. For this reason assembly language need no longer be feared by the average programmer; instead, it becomes a language within the grasp of everyone.

MICRO

Supermicros

(continued from page 45)

and BASIC programs run four times faster than on IBM's Personal Computer. It is as fast to program in high level compiler languages as using interpreters on 8-bit machines. Our 6502 assembly language programming productivity doubled.

With an unexpanded, 256K SAGE II, you can plug in your own 64K bit dynamic RAM chips for 512K bytes and your own second Mitsubishi floppy disk drive; sockets, cables and connectors are provided with the unexpanded machine. One hundred and fifty nanosecond, 64K bit RAM chips cost about six dollars each at the present, and 36 chips make up 256K of RAM memory.

The video display and keyboard aren't built-in on the SAGE, unlike the Apple II; a separate RS232 serial terminal is required. However, not having a built-in display and keyboard can be advantageous, because the user only pays for what he needs. Separate 19.2K baud serial terminals are also faster than most built-in hi-res bit-mapped displays. This is due to the dedicated CPU in the terminal that has nothing else to do but update the screen while bit mapped displays are usually updated by the main CPU. (Multiprocessing if you will). It has 640K on each floppy disk drive, 512K of parity RAM and 24 bit address, 16 bit data bus, expansion connectors. It comes with a built-in Centronics parallel printer port, an IEEE-488 port and two RS-232 serial ports, one which is used with the terminal, the other already set up for a modem. Options include hard disk up to forty megabytes and a six-user system with 1 megabyte RAM.

Several other operating systems will run on the SAGE, including CP/M 68K, Mirage, PDOS, BOS/5, MBOS/5 and Idris (a UNIX-like operating system). Languages that run under the standard and optional operating systems are several versions of Fortrans, BASIC, ADA, Forth, Cobol, Microcobol, APL, Modula II and several "C"s.

Here's the most serious advice to anyone contemplating writing software: write it in assembly language for the 68000. The 68000 and its derivatives will become the de facto standard microprocessors for at least the next ten years, despite IBM's temporary lead with the 8088.

MICRO

Useful Functions

by Paul Garrison

Editor's Note: The following program is given in its entirety to illustrate one way of setting up a program to easily access various defined functions. In the next two issues, programs 2 and 3 will be published in their entirety. We invite you to send in any defined functions you may be using that are not mentioned. The submissions we receive will be collected and published in a future issue.

Save time and mathematical aggravation with this compilation of defined functions in a very friendly program.

Many of us, depending on what we do for a living, find that we must use a variety of arithmetic formulas, and more often than not we're faced with the task of looking them up in some book or other research material, after which we must key them into our computer or calculator making sure that they're exactly correct. That can, on occasion, become quite a task when the formula involved includes a half dozen pairs of parentheses or other complicated combinations of fixed and variable values. A typical example of such a formula is shown below. Such lengthy arithmetic expressions simply invite errors.

The programs that make up the main portion of this article are designed to simplify the task. They are made up

```
61 DEF FNDENALT(PA,F)=(145426*
(1-(((288.15-PA*.ØØ1981)/288.15)
↑5.2563/((273.15+F)/288.15))
↑.235))
```

of 60-odd user-defined functions representing all of the arithmetic expressions that I have ever needed in the more or less technical writing that I have done.

In addition, all three include a group of three subroutines and an END line (lines 130 through 160) that I automatically put in all the programs I write. My reason for dividing the functions into three separate programs is based on the need to keep them within a 48K limit, or rather the 14-plus K limit that is available with a 48K RAM when MBASIC is loaded into the computer.

The programs consist of the actual functions (lines 1 to 99), where the line numbers are not duplicated in the three programs in order to be able to merge portions of the programs into new programs without getting involved with a confusion of line numbers,the subroutines (lines 130 thru 160), a menu (lines 200 thru 400) plus a means of using all of the defined functions in order to perform a given calculation with optional variables. The three programs are recorded on a disk that also contains the CP/M system, MBASIC and the CP/M PIP program which simplifies the task of copying them onto another disk where they can then be merged with a new program. Since it is unlikely that all of the functions will be used in such a program, it is then a simple matter to delete those functions and other material not applicable to the program being written.

The programs are written in the Apple version of Microsoft BASIC-80 (MBASIC), using the WordStar wordprocessor. The changes that must be made in order to translate them into other versions of BASIC are described below.

Other BASIC Dialects

VARIABLE NAMES: BASIC-80, TI BASIC, TI EXTENDED BASIC, Atari BASIC and some other versions will recognize 40 or more characters in a variable name, while Applesoft, TRS-80 and several others recognize only the first two characters (plus the $ in case of string variables). Therefore, variable names that exceed two characters may have to be examined in order to avoid inadvertant duplication. For instance, Applesoft and TRS-80 would look at BOLD and BOLT and read both as BO.

MULTIPLE STATEMENTS PER LINE: In most versions of BASIC, multiple statements on one line, separated by a colon (:) are permitted. In TI BASIC, multiple statements per line are not accepted. In TI EXTENDED BASIC, multiple statements must be separated by two colons(::).

INPUT WITH PROMPT STATEMENT: Most versions of BASIC accept INPUT followed by a prompt statement in quotation marks. In BASIC-80, the prompt statement may be followed by a semicolon, resulting in a displayed question mark (?), or by a comma eliminatingthe question mark. In Applesoft and TRS-80, the prompt statement must always be followed by a semicolon. In both versions of TI BASIC, the prompt statement must be followed by a colon. Atari does not permit a prompt line after INPUT. Instead, the prompt must be used as a PRINT line, followed by INPUT and the variable(s).

DEFINED FUNCTIONS: BASIC-80 uses DEF FNABC(X) with no space between FN and the function name ABC(X). In Applesoft, it can be typed without a space, but the computer will insert a space automatically, resulting in FN ABC(X). The two TI BASICs do not use FN. Instead DEF ABC(X) is used to define ABC. In the reference books for TRS-80 and Atari, I have been unable to find the DEFine command. The way to get around that is to simply assign an arithmetic expression to a variable name; ABC = 1/A would, when ABC is PRINTed, display the reciprocal of the value assigned to the numeric variable A. In that case no variables in parentheses can be used because the computer would recognize that as an array, and would respond with a SUBSCRIPT OUT OF RANGE error message if the variable in parentheses exceeds the maximum allowable number and no prior DIM statement was encountered.

TO CLEAR THE SCREEN: BASIC-80 and Applesoft use HOME to clear the screen. TRS 80 uses CLS for the purpose. With computers that do not include a clear-screen command, use FOR X = 1 TO L:PRINT:NEXT X where L is the number of lines displayed on the screen. The two TI BASICs use CALL CLEAR for that purpose.

TAB(X) and VTAB(X): In the TI BASICs, the TAB(X) statement must be followed by a semicolon. In some versions of BASIC TAB and/or VTAB are not available. In that case, spaces within parentheses can be used to effect the TAB position and FOR X = 1 TO Z:PRINT:NEXT X can be used to move the text to a given vertical position on the screen, represented by the value of Z.

There are other differences between the versions of BASIC used by different computer makes and models, but these are the only ones used in the programs reproduced here.

PROGRAM #1

This program contains the mathematical formulas for all versions of SINE, TANGENT, and SECANT that are not built in functions (such as SIN(X), COS(X), TAN(X) and ATN(X)) automatically available on all microcomputers. In addition, it contains conversions of degrees to radians and vice versa which are frequently needed in conjunction with the others.

Lines 2 and 3 assign standard values to PI and RAD. Lines 16 through 37 use the DEF FN statement to create the user-defined functions. Lines 100 through 160 are the lines that I use at the beginning of all my programs. Lines 200 through 400 contain the menu that allows you to use any of the defined functions to perform a given calculation, using your own variables. Line 420 sends the computer to the appropriate line number based on the selection made from the menu. And lines 690 through 1560 are used to perform the different calculations.

PROGRAM #2

This program includes the formulas for trigonometric ratios, two formulas dealing with matters related to aviation (the effect of wind on ground speed and density altitude), the formulas for converting temperatures from Fahrenheit to Celsius and vice versa, plus the formulas that comprise Ohm's Law and determine the resistance factor of electrical wires, and finally the formula that determines future values based on compound interest, present value and the time span to be examined. The structure of the program is identical to the one described above. See editor's note.

PROGRAM #3

This program contains a variety of formulas, such as those used to determine the lesser, greater or average value of two variables, rounding off figures to a given number of decimals, polar-to-rectangular and rectangular-to-polar conversions, figuring roots of any variables (square root, cube root and so on), determining the reciprocal of any number, and determining the surface areas and volumes of cubes, rectangular shapes, spheres, pyramids and cylinders. Beyond that the program is structured like the others, except that the menu is at the end (lines 2000 and up). See editor's note. MICRO

Listing for Program 1

```
1 REM FUNCTIONS (DELETE THOSE NOT USED IN A PROGRAM)
2 PI=3.14159
3 RAD=57.2958
16 DEF FNARCSIN(A)=ATN(A/SQR(-A*A+1)):                  REM ARCSINE
17 DEF FNSINH(A)=(EXP(A)-EXP(-A))/2:                    REM HYPERBOLIC SINE
18 DEF FNARCCOS(A)=-ATN(A/SQR(-A*A+1))+1.5708:          REM ARCCOSINE
19 DEF FNCOSH(A)=(EXP(A)+EXP(-A))/2:                    REM HYPERBOLIC COSINE
20 DEF FNCOT(A)=1/TAN(A):                               REM COTANGENT
21 DEF FNARCCOT(A)=ATN(A)+1.5708:                       REM ARCCOTANGENT
22 DEF FNTANH(A)=EXP(-A)/(EXP(A)+EXP(-A))*2+1:          REM HYPERBOLIC TANGENT
23 DEF FNCOTH(A)=EXP(-A)/(EXP(A)-EXP(-A))*2+1:          REM HYPERBOLIC COTANGENT
24 DEF FNSEC(A)=1/COS(A):                               REM SECANT
25 DEF FNCSC(A)=1/SIN(A):                               REM COSECANT
26 DEF FNARCSEC(A)=ATN(A/SQR(A*A-1))+SGN(SGN(A)-1)*1.5708:REM ARCSECANT
27 DEF FNARCCSC(A)=ATN(A/SQR(A*A-1))+(SGN(A)-1)*1.5708:  REM ARCCOSECANT
28 DEF FNSECH(A)=2/(EXP(A)+EXP(-A)):                    REM HYPERBOLIC SECANT
29 DEF FNARCSINH(A)=LOG(A+SQR(A*A+1)):                  REM HYPERBOLIC ARCSINE
30 DEF FNARCCOSH(A)=LOG(A+SQR(A*A+1)):                  REM HYPERBOLIC ARCCOSINE
31 DEF FNARCTANH(A)=LOG((1+A)/1-A)/2:                   REM HYPERBOLIC ARCTANGENT
32 DEF FNARCSECH(A)=LOG((SQR(-A*A+1)+1)/A):             REM HYPERBOLIC ARCSECANT
33 DEF FNARCCOTH(A)=LOG((A+1)/(A-1))/2:                 REM HYPERBOLIC ARCCOTANGENT
34 DEF FNARCSCSH(A)=LOG((SGN(A)*SQR(A*A+1)+1)/A):       REM HYPERBOLIC ARCCOSECANT
36 DEF FNDEG(A)=A*(PI/180):                             REM DEGREES TO RADIANS
37 DEF FNRAD(A)=A/(PI/180):                             REM RADIANS TO DEGREES
120 GOTO 200
130 ?"--------------------------------------------------------------------":RETURN
140 HOME:VTAB(10):RETURN
150 ?:INPUT "Press >RETURN<  (Q to quit)   ",R$
155 IF R$="Q" THEN 160 ELSE RETURN
160 GOSUB 140:GOSUB 130:?TAB(33)"End.":GOSUB 130:END
190                                                      REM TESTING FUNCTIONS
200 GOSUB 140:?"Menu:":GOSUB 130
210 ?1,"Arcsine"
220 ?2,"Hyperbolic sine"
230 ?3,"Arccosine"
240 ?4,"Hyperbolic cosine"
250 ?5,"Cotangent"
260 ?6,"Arccotangent"
270 ?7,"Hyperbolic tangent"
280 ?8,"Hyperbolic cotangent"
290 ?9,"Secant"
300 ?10,"Cosecant"
310 ?11,"Arcsecant"
320 ?12,"Arccosecant":GOSUB 130
322 ?"To choose one of the above, press >RETURN< "
324 INPUT "To see other choices, press > Y <    ",Z$
326 IF Z$="Y" THEN GOSUB 130:GOTO 330 ELSE GOSUB 130:GOTO 410
330 ?13,"Hyperbolic secant"
340 ?14,"Hyperbolic arcsine"
350 ?15,"Hyperbolic arccosine"
360 ?16,"Hyperbolic arctangent"
370 ?17,"Hyperbolic cosecant"
380 ?18,"Hyperbolic arccotangent"
390 ?19,"Hyperbolic arccosecant":GOSUB 130
392 ?20,"Convert degrees to radians"
394 ?21,"Convert radians to degrees":GOSUB 130
400 ?22,"Exit program":GOSUB 130
410 INPUT "Which?          ",WHICH:GOSUB 140
420 ON WHICH GOTO 690,730,770,810,850,890,930,970,1010,1050,1090,1130,1170,1210,
   1250,1290,1330,1370,1410,1490,1530,160
690 ?"Find the arcsine of a number":GOSUB 130
700 INPUT "Enter any number                    ",A
710 X=FNARCSIN(A):GOSUB 130
720 PRINT "The arcsine of ";A;" is             ";X:GOSUB 150:GOTO 200
730 ?"Find the hyperbolic sine of a number":GOSUB 130
740 INPUT "Enter any number                    ",A
750 X=FNSINH(A):GOSUB 130
760 PRINT "The hyperbolic sine of ";A;" is     ";X:GOSUB 150:GOTO 200
770 ?"Find the arccosine of a number":GOSUB 130
780 INPUT "Enter any number                    ",A
790 X=FNARCCOS(A):GOSUB 130
800 PRINT "The arccosine of ";A;" is           ";X:GOSUB 150:GOTO 200
```

Listing 1 *(continued)*

```
810 ?"Find the hyperbolic cosine of a number":GOSUB 130
820 INPUT "Enter any number                         ",A
830 X=FNCOSH(A):GOSUB 130
840 PRINT "The hyperbolic cosine of ";A;" is      ";X:GOSUB 150:GOTO 200
850 ?"Find the cotangent of a number":GOSUB 130
860 INPUT "Enter any number                         ",A
870 X=FNCOT(A):GOSUB 130
880 PRINT "The cotangent of ";A;" is                ";X:GOSUB 150:GOTO 200
890 ?"Find the arccotangent of a number":GOSUB 130
900 INPUT "Enter any number                         ",A
910 X=FNARCCOT(A):GOSUB 130
920 PRINT "The arccotangent of ";A;" is             ";X:GOSUB 150:GOTO 200
930 ?"Find the hyperbolic tangent of a number":GOSUB 130
940 INPUT "Enter any number                         ",A
950 X=FNTANH(A):GOSUB 130
960 PRINT "The hyperbolic tangent of ";A;" is     ";X:GOSUB 150:GOTO 200
970 ?"Find the hyperbolic cotangent of a number":GOSUB 130
980 INPUT "Enter any number                         ",A
990 X=FNCOTH(A):GOSUB 130
1000 PRINT "The hyperbolic cotangent of ";A;" is  ";X:GOSUB 150:GOTO 200
1010 ?"Find the secant of a number":GOSUB 130
1020 INPUT "Enter any number                        ",A
1030 X=FNSEC(A):GOSUB 130
1040 PRINT "The secant of ";A;" is                  ";X:GOSUB 150:GOTO 200
1050 ?"Find the cosecant of a number":GOSUB 130
1060 INPUT "Enter any number                        ",A
1070 X=FNCSC(A):GOSUB 130
1080 PRINT "The cosecant of ";A;" is                ";X:GOSUB 150:GOTO 200
1090 ?"Find the arcsecant of a number":GOSUB 130
1100 INPUT "Enter any number                        ",A
1110 X=FNARCSEC(A):GOSUB 130
1120 PRINT "The arcsecant of ";A;" is               ";X:GOSUB 150:GOTO 200
1130 ?"Find the arccosecant of a number":GOSUB 130
1140 INPUT "Enter any number                        ",A
1150 X=FNARCCSC(A):GOSUB 130
1160 PRINT "The arccosecant of ";A;" is             ";X:GOSUB 150:GOTO 200
1170 ?"Find the hyperbolic secant of a number":GOSUB 130
1180 INPUT "Enter any number                        ",A
1190 X=FNSECH(A):GOSUB 130
1200 PRINT "The hyperbolic secant of ";A;" is      ";X:GOSUB 150:GOTO 200
1210 ?"Find the hyperbolic arcsine of a number":GOSUB 130
1220 INPUT "Enter any number                        ",A
1230 X=FNARCSINH(A):GOSUB 130
1240 PRINT "The hyperbolic arcsine of ";A;" is     ";X:GOSUB 150:GOTO 200
1250 ?"Find the hyperbolic arccosine of a number":GOSUB 130
1260 INPUT "Enter any number                        ",A
1270 X=FNARCCOSH(A):GOSUB 130
1280 PRINT "The hyperbolic arccosine of ";A;" is  ";X:GOSUB 150:GOTO 200
1290 ?"Find the hyperbolic arctangent of a number":GOSUB 130
1300 INPUT "Enter any number                        ",A
1310 X=FNARCTANH(A):GOSUB 130
1320 PRINT "The hyperbolic arctangent of ";A;" is ";X:GOSUB 150:GOTO 200
1330 ?"Find the hyperbolic arcsecant of a number":GOSUB 130
1340 INPUT "Enter any number                        ",A
1350 X=FNARCSECH(A):GOSUB 130
1360 PRINT "The hyperbolic arcsecant of ";A;" is  ";X:GOSUB 150:GOTO 200
1370 ?"Find the hyperbolic arccotangent of a number":GOSUB 130
1380 INPUT "Enter any number                        ",A
1390 X=FNARCCOTH(A):GOSUB 130
1400 PRINT "The hyperbolic arccotangent of ";A;" is ";X:GOSUB 150:GOTO 200
1410 ?"Find the hyperbolic arccosecant of a number":GOSUB 130
1420 INPUT "Enter any number                        ",A
1430 X=FNARCSCSH(A):GOSUB 130
1440 PRINT "The hyperbolic arccosecant of ";A;" is ";X:GOSUB 150:GOTO 200
1490 ?"Convert degrees to radians":GOSUB 130
1500 INPUT "Enter number of degrees                 ",A
1510 X=FNDEG(A):GOSUB 130
1520 PRINT A;" degrees equal ";X;" radians":GOSUB 150:GOTO 200
1530 ?"Convert radians to degrees":GOSUB 130
1540 INPUT "Enter number of radians                 ",A
1550 X=FNRAD(A):GOSUB 130
1560 ?A;" radians equal ";X;" degrees":GOSUB 150:GOTO 200
```

Apple IIe Supplement to What's Where in the Apple

by Phil Daley

A.1 Overview

The latest Apple II, called the "//e" for "enhanced", has several features added that make it more standard and versatile. The keyboard has been improved and will now generate all 128 ASCII key codes, including screen display of lower case. The RESET key now requires pressing the CONTROL key simultaneously and rebooting can be accomplished by pressing CTRL-OPEN APPLE-RESET, saving wear and tear on the on/off switch, always a weak point. A CTRL-CLOSED APPLE RESET initiates a built-in self-test. The screen display has been improved to allow either 40 or 80 column display under software control. There is also a full cursor control in all four directions. The 16K language card has been made a built-in feature and slot 0 has been eliminated. International versions are available for European and Asian buyers with switchable character sets.

Despite all these additional features, compatability was kept with most of the previous software. All of the standard monitor entry points were preserved so that, unless software uses undocumented monitor entries, it should run on the //e. The only other problem that might arise is the utilization of one formerly unused page zero location. A program that used that location will probably not function properly on the new Apple.

Another new feature is the addition of a 64K expansion available as an enhanced 80 column card, which will make additional memory available to sophisticated programs such as Visicalc.

A.2 A Third Apple Monitor

There is now a third major version of the Apple monitor to go along with the Auto-Start and (old) System monitors. While all of the documented entry points remain the same, most of the routines jump to the new ROM in the $C100-$CFFF range. These new routines check on the availability and status of 80 column and extended 80 column cards, and use this additional hardware for enhanced displays and cursor control, when available.

The major differences between the II+ and the //e are as follows:

a) RESET, OPEN APPLE and CLOSED APPLE keys: The Control key must now be pressed to initiate the RESET cycle. This will eliminate accidental RESETs as the keys are on opposite sides of the keyboard. The APPLE keys are paddle button extensions to the keyboard and can be used in conjunction with the RESET cycle to initiate the self diagnostic tests (CLOSED) or power-on reboot (OPEN).

b) EDITING: In addition to the I, J, K, and L diamond cursor control pattern, there are four arrow keys that can also be used to move the cursor on the screen. Pressing ESC to enter the editing mode changes the cursor to an inverse " + " to indicate editing mode. Additional commands are also available. ESC-R enters upper-case restrict mode, which allows only upper-case letters during keyboard entry except after typing a """, when both upper and lower case are allowed for PRINT statement. Typing another """ returns to upper-case only. ESC-T exits this mode. ESC-4 displays a 40 column screen similar to the II+, while ESC-8 shifts to the new 80 column screen display. ESC CTRL-Q exits the new made entirely, returning to the old 40 column display, and turning off the 80 column card.

A.3 The New Display

In order to maintain compatability with the old II and II+, it was necessary to design a screen display that utilized the old screen memory ($400-$7FF). This was insufficient for 80 column display, so Apple designed an 80 column card with its own memory mapped into the same addresses. The hardware alternates its scans from one set of memory to the other when in 80 column mode. Characters are stored alternating from one address to the next, with all the odd screen locations in main memory and all the even ones on the auxiliary card.

There are routines in the new monitor areas that can convert an 80 column screen to 40 by moving the alternate characters to the main board and throwing away the last 40 characters in each column. The opposite switch is accomplished by a similar move to the auxiliary card, using only the leftmost 40 columns for the characters previously on the screen.

A.4 Hardware Locations

On the older Apples, the addresses \$C000-\$C00F were equivalent addresses and were only partly decoded by the hardware. This meant that reading any of those would yield the same result (reading the keyboard), which was also true of \$C010-\$C01F (clearing the keyboard strobe). These addresses are now fully decoded and provide a set of soft switches/status indicators for the new 80 column card and extended 80 column card (with 64K memory expansion).

The switches include options to read and/or write either the main board locations or the auxiliary card locations, to set the standard zero page and system stack (main board) or the alternate zero page and system stack (auxiliary card), to turn on or off the \$CX00 ROMs, to enable or disable the 80 column display, and to turn on the normal or alternate character sets (normal has upper case flash instead of lower case inverse).

Additionally, there are a group of locations that can be read to determine the current switch settings so that any program changing the switches can save the current settings and restore them at the end. States that can be determined include READ/WRITE status, language card bank status, 80 column status, page status, and text mode.

A.5 Software Status

Apple has always reserved some unused locations in the text page RAM as scratch memory for the 7 hardware slots (1-7). Several of these locations are now permanently assigned to the new 80 column cards, when they are in use, and are used to store the current cursor location, I/O status, and BASL/BASH in Pascal.

One particular location (\$4FB) is the software MODE status. Each bit is indicative of the current state of operations: BASIC/Pascal, interrupts set/cleared, Pascal 1.0/1.1, normal/inverse video, GOTOXY in progress/not in progress, upper case restrict/literal mode, BASIC input/print, and ESC-R active/inactive.

These locations enable a program to determine the current state of the machine more easily than before, and make it simpler to utilize the new hardware configurations in programming.

A.6 Programming Considerations

The standard Applesoft GET and INPUT (and associated monitor routine KEYIN) were not designed to work with an 80 column display and using them while in 80 column mode can cause loss of data or erasure of program in memory, but this can be overcome by a routine explained in Appendix E of the new Applesoft Tutorial. Reading the keyboard directly (\$C000) functions the same as before.

Do not assume an Apple //e or 80 column card when writing programs; one of the first routines should check for the type of machine being used. Apple supplies a program that will do this on "The Applesoft Sampler"; and Call A.P.P.L.E. has also published a routine for this purpose. HTAB will not function beyond the 40th column. While POKE 36,POS works most of the time, Apple recommends POKE 1403,POS (0-79) for the //e. This routine will not work at all for an old Apple.

It is the programmer's responsibilty to turn off the 80 column card at the end of a program. Do not quit the card with the cursor beyond the 39th column, as this can cause unpredictable results including program erasure. In case of accidently executing this command, pressing RETURN immediately will usually recover the cursor to the left margin. It is also necessary to turn the 80 column card off before sending output to printers, modems, etc.

VTAB no longer works when a window is set (by POKing 32,33 etc.). The solution is to VTAB to the location -1, and then do a PRINT prior to PRINTing the actual data. This causes the firmware to recognise the new VTAB location.

These cautions are a small price to pay for the increased versatility and flexability of the new Apple //e.

Editor's Note: This material is intended to be used in conjunction with the original version of What's Where in the Apple which did not contain Apple IIe material. The 1984 edition of WWA is now available for \$19.95 from your local bookstore, computer store, or by mail from MICRO, P.O. Box 6502, Chelmsford, MA 01824 (add \$2 shipping)

```
HEX LOCN (DEC LOCN) [NAME]  \USE-TYPE\ - DESCRIPTION
------------------------------------------------------------------------------------------

There is 1 page 0 location that was not formerly used which is now used.

$1F (31) [YSAV1] \P1\              Temporary storage for the Y register


There are several locations in the text page that are storage for permanent data in these unused screen locations.
Any routine which sets page 2 must restore page 1 so that these data may be accessed.


$478 (1144) [TEMP1] \P1\           A temporary storage location
$47B (1147) [OLDCH] \P1\           Old CH set for user
$4FB (1275) [MODE] \P1\            Current operating mode acording to the following bits:

                                   Bit 0 Off Normal mode (Pascal)
                                   Bit 0 On  Transparent mode (Pascal)
                                   Bit 0 Off Caller set interrupts (BASIC)
                                   Bit 0 On  Caller cleared interrupts (BASIC)
                                   Bit 1 Off Pascal 1.1 F/W active
                                   Bit 1 On  Pascal 1.0 Interface
                                   Bit 2 Off Normal video (Pascal)
                                   Bit 2 On  Inverse video (Pascal)
                                   Bit 3 Off GOTOXY not in progress
                                   Bit 3 On  GOTOXY in progress
                                   Bit 4 Off Upper case restrict mode
                                   Bit 4 On  Literal upper/lower case mode
                                   Bit 5 Off Current language is BASIC
                                   Bit 5 On  Current language is Pascal
                                   Bit 6 Off BASIC PRINT
                                   Bit 6 On  BASIC INPUT
                                   Bit 7 Off ESC-R inactive
                                   Bit 7 On  ESC-R active

$57B (1403) [OURCH] \P1\           80 column CH
$5FB (1531) [OURCV] \P1\           Cursor vertical
$67B (1659) [CHAR] \P1\            In/Out character
$6FB (1787) [XCOORD] \P1\          X coordinate in GOTOXY routine
$77B (1915) [OLDBASL] \P1\         Pascal saved BASL
$7FB (2043) [OLDBASH] \P1\         Pascal saved BASH
```

```
HEX LOCN (DEC LOCN) [NAME]  \USE-TYPE\ - DESCRIPTION
-----------------------------------------------------------------------------------------------------------

$C000-$C01F (49152-49183) \H\       Hardware locations/switches
$C000 (49152) [CLR80COL] \H1\       Disable 80 column store
$C001 (49153) [SET80COL] \H1\       Enable 80 column store
$C002 (49154) [RDMAINRAM] \H1\      Read RAM on mainboard
$C003 (49155) [RDCARDRAM] \H1\      Read RAM on card
$C004 (49156) [WRMAINRAM] \H1\      Write RAM on mainboard
$C005 (49157) [WRCARDRAM] \H1\      Write RAM on card
$C007 (49159) [SETINTCXROM] \H1\    Set internal CX00 ROM
$C008 (49160) [SETSTDZP] \H1\       Set standard zero page/stack
$C009 (49161) [SETALTZP] \H1\       Set alternate zero page/stack
$C00B (49163) [SETSLOTC3ROM] \H1\   Enable C300 slot ROM
$C00C (49164) [CLR80VID] \H1\       Disable 80 column video
$C00D (49165) [SET80VID] \H1\       Enable 80 column video
$C00E (49166) [CLRALTCHAR] \H1\     Normal lower case, flash upper case
$C00F (49167) [SETALTCHAR] \H1\     Normal/inverse lower case, no flash
$C011 (49169) [RDLCBNK2] \H1\       Reads language card bank 2
$C012 (49170) [RDLCRAM] \H1\        Reads language card RAM enable
$C013 (49171) [RDRAMRD] \H1\        Reads RAMREAD state
$C014 (49172) [RDRAMWRT] \H1\       Reads BANKWRT state
$C018 (49176) [RD80COL] \H1\        Reads SET80COL
$C019 (49177) [RDVBLBAR] \H1\       Reads VBL signal
$C01A (49178) [RDTEXT] \H1\         Reads Text mode
$C01C (49180) [RDPAGE2] \H1\        Reads page 1/2 status
$C01F (49183) [RD80VID] \H1\        Reads SET80VID
$C100-$CFFF (49408-53247) [CX00ROM] \SB\ A new set of subroutines to handle the 80 column card and auxilliary
                                    memory in slot 3. It is entered from the GOTOCX subroutine in the F800 ROM which
                                    sets interrupts, turns on the CX00 ROMs, and JMPs to C100. Function code is in
                                    Y reg. Note: "B." routines are the new way. "F." routines are the old way.
                                    Stack has status of bank and IRQ. Uses A,Y registers.
                                    Function Codes:
                                    0      CLREOP
                                    1      HOME
                                    2      SCROLL
                                    3      CLREOL
                                    4      CLEOLZ
                                    5      INIT & RESET
                                    6      KEYIN
                                    7      Fix ESCape Character
                                    8      SETWND
                                    If there is a card in the slot then the new video routines are used, since the
                                    screen hole locations belong to the card. Otherwise the F8 ROM routines are
                                    duplicated to avoid slot 3 interference with another type of interface.
$C100 (49408) [B.FUNC] \SE\         Entry point for all routines with code in Y.Check first for KEYIN Y=6
$C107 (49415) [B.FUNCNK] \SE\       Check for ESCape-fix Y=7
$C10E (49422) [B.FUNCNE] \SE\       Test for card. If present, use the new routines, if not, old routines

-----------------------------------------------------------------------------------------------------------
```

```
$C11F (49439) [B.OLDFUNC] \SE\     Pushes $C1 on stack, and low byte address of the function -1 by looking up in
                                   F.TABLE indexed by Y. Then does fake RTS to routine.
$C129 (49449) [F.CLREOP] \SE\      Monitor S/R to clear from the cursor to the end of page.
$C143 (49475) [F.HOME] \SE\        Clear scroll window to blanks. Set cursor to top left corner.
$C14D (49485) [F.SCROLL] \SE\      Monitor S/R to scroll up one line.
$C17D (49533) [F.CLREOL] \SE\      Monitor S/R to clear to end of line.
$C18A (49546) [F.SETWND] \SE\      Monitor S/R to set normal low-resolution graphics window, cursor bottom left.
$C19C (49564) [F.CLEOLZ] \SE\      Monitor S/R to clear entire line.
$C1A1 (49569) [F.GORET] \L\        Exit routine to F.RETURN
$C1A4 (49572) [B.FUNCO] \SE\       Entry point to new routines. Sets the IRQ mode and screen holes, Y reg.
$C1CD (49613) [B.SCROLL] \SE\      Entry point for monitor routine to scroll up one line
$C1D3 (49619) [B.CLREOL] \SE\      Entry point for monitor routine to clear to end of line
$C1D9 (49625) [B.CLEOLZ] \SE\      Entry point for monitor routine to clear entire line
$C1E1 (49633) [B.CLREOP] \SE\      Entry point for monitor routine to clear to end of page
$C1E7 (49639) [B.SETWND] \SE\      Entry point for monitor routine to set text window
$C1EA (49642) [B.RESET] \SE\       Entry point for monitor routine to reset entire system
$C1ED (49645) [B.HOME] \SE\        Monitor S/R to clear the text page and put cursor in upper left corner
$C1FF (49663) [B.VECTOR] \SE\      Monitor S/R to check on 80 col use and get current Cursor Horizontal position (CH)
$C20E (49678) [B.GETCH] \SE\       Save CH in screenhole
$C211 (49681) [B.FUNC1] \SE\       Pushes $C1 on stack, and low byte address of the function -1 by looking up in
                                   B.TABLE indexed by Y. Then does fake RTS to routine.
$C219 (49689) [B.SETWNDX] \SE\     Monitor S/R to set normal text window 40/80 columns
$C234 (49716) [B.RESETX] \SE\      Monitor routine to reset system, checks for "Apple" keys for cold start, else does
                                   warm restart without diagnostics, blasts memory from BFXX down to stack, checks
                                   80 col board to see if CX ROM needs resetting, and returns
$C261 (49761) [DIAGS] \SE\         Entry point for monitor S/R diagnostics
$C26E (49774) [B.ESCFIX] \SE\      Monitor S/R to map i,j,k,m and <-,^,->, and V into I,J,K,M for cursor movement
                                   Returns with old form of character in A.
$C280 (49792) [ESCIN] \P4\         Table of arrow keys
$C284 (49796) [ESCOUT] \P4\        "J,K,M,I" translations for arrows
$C288 (49800) [B.KEYIN] \SE\       Monitor routine to read a key with new additions to save CX bank status, check
                                   interrupt status, put new cursor ASC"$FF" on screen, JSR to KEYDLY (old RDKEY),
                                   restore the original screen character, put the new character in A reg., clear the
                                   keyboard strobe and return to caller.
$C2C6 (49862) [KEYDLY] \SE\        Monitor routine to get a key from KBD, also checking interrupts, and still
                                   incrementing RNDL and RNDH, the random locations
$C2EB (49899) [F.RETURN] \SE\      Monitor routine to exit from CX ROM routines either leaving I/O disabled or
                                   enabling it if it was on entry
$C300 (49920) [BASICINT] \SE\      Sets INIT Flag (V) and branches to BASIC I/O entry point
$C307 (49927) [BASICOUT] \SE\      Clears INIT Flag (V) and branches to BASIC I/O entry point
$C30B (49931) [PASFPT] \P6\        Pascal 1.1 firmware protocol table
$C311 (49937) [128KJMP] \P6\       Jump table for 128K support routines
$C317 (49943) [BASICENT] \SE\      BASIC I/O entry point, saves CHAR, A, Y, X, and P, pulls P from stack, checks IRQ
                                   status, and sets appropriately.
```

```
HEX LOCN (DEC LOCN) [NAME]  \USE-TYPE\ - DESCRIPTION
------------------------------------------------------------------------------------------------------

$C336 (49974) [BASICENT2] \SE\        Turns off any slots using C8 area, sets C8SLOT to $C3, checks INIT flag, and jumps
                                      to warm or cold BASIC in C8 ROM
$C34B-$C362 (49995-50018) [PJUMPS]    Pascal jump table
$C34B (49995) [JPINIT] \SE\           Pascal INIT
$C351 (50001) [JPREAD] \SE\           Pascal READ
$C357 (50007) [JPWRITE] \SE\          Pascal WRITE
$C35D (50013) [JPSTAT] \SE\           Pascal STATUS
$C363 (50019) [MOVE] \SE\             Monitor S/R to move memory across memory banks.   Call with A1 = Source start,
                                      A2 = Source end, A4 = Destination start, Carry set for Main to Card,
                                      Carry clear for Card to Main.
$C3B0 (50096) [XFER] \SE\             Transfer program control from main board to card or vice versa. $3ED-$3EE is
                                      address to be executed upon transfer, carry set means transfer to card, carry
                                      clear means transfer to main board, V flag clear means use standard zero
                                      page/stack, V flag set means use alternate zero page/stack.
                                      Also uses $3ED-$3EE in destination bank.
                                      Enter via JMP not JSR.
$C3EB (50155) [SETC8] \SE\            Setup IRQ C800 protocol. Stores $C3 in C8SLOT.

$C800 (51200) [PINIT1] \SE\           Pascal 1.0 init
$C803 (51203) [BASICINIT] \SE\        Checks the F8 ROM version, if not //e, copies ROM to RAM Card, and checks again,
                                      if still not good, hangs the system.
$C816 (51222) [BINIT1] \SE\           Set up BASIC I/O in CSW and KSW to point to BASICENT in the C3 ROM and set text or
                                      graphics windows
$C84B (51272) [PREAD1.0]              Pascal 1.0 input hook
$C850 (51280) [BINIT2] \L\            Check for 80 column mode and enable, if so
$C85D (51293) [CLEARIT] \L\           Monitor routine to set lower case mode, clear screen and clears carry
$C866 (51302) [C8BASIC] \L\           Monitor routine to check mode and set 80 column store in case Integer BASIC cleared
                                      Also rounds WNDWDTH to next lower even, if odd in 80 column mode.
$C874 (51316) [C8B2] \L\              Monitor routine to check current CH and store it if different from OLDCH
$C87E (51326) [C8B3] \L\              Monitor routine to check RAM card for correct version and, if not, recopy the
                                      F8ROM to RAM card , check again and hang if not correct.
$C890 (51344) [C8B4] \L\              Monitor routine to check carry, on clear-print a character, set-input a character
$C896 (51350) [BOUT] \SE\             Monitor S/R to set MODE to BASIC printing, falls through to BPRINT
$C8A1 (51361) [BPRINT] \SE\           Monitor S/R to output character in CHAR, checks for CTRL-S, clears high bit, checks
                                      for CTRL chars, if it is, process and return, if not, fall through to BPNCTL.
$C8CC (51404) [BPNCTL] \SE\           Monitor S/R to reload CHAR (to get 8th bit, and print the char on the screen,
                                      Increments cursor horizontal and scrolls, if necessary
$C8E2 (51426) [BIORET] \L\            Monitor routine to store cursor position, restore X, Y, and A and return to BASIC
$C8F6 (51446) [BINPUT] \SE\           Monitor routine to set MODE to BASIC input, get the cursor position, and CHAR
$C905 (51461) [B.INPUT] \SE\          Monitor routine to inverse char at current position, get a char from the keyboard,
                                      remove cursor, and process char, including ESCapes. If not ESC then JMP to NOESC.
```

```
HEX LOCN (DEC LOCN) [NAME]  \USE-TYPE\ - DESCRIPTION
------------------------------------------------------------------------------------------------
$C918 (51480) [ESCAPING] \SE\       Monitor routine to process ESCape command sequences. The commands are:
                                    @ - Home and Clear screen
                                    E - Clear to end of line
                                    F - Clear to end of page
                                    A,K,-> - Cursor right
                                    B,J,<- - Cursor left
                                    C,M,V - Cursor down
                                    D,I,^ - Cursor up
                                    R - Restrict to uppercase
                                    T - Turn off Esc-R
                                    4 - Go to 40 column mode
                                    8 - Go to 80 column mode
                                    CTRL-Q - Quit new routines. (PR#0/IN#0)

                                    Places ESCape cursor on screen, GETs a command key, puts lower case into upper,
                                    checks the ESCTAB for a valid character. If the char is there, load A with the Y
                                    index into ESCCHAR, and "print" the control character, if its not, check for "T",
                                    "R" and "CTRL-Q" special functions and process, if its not, return to caller.
                                    If the ESCCHAR entry has the high bit set, return to ECSAPING, otherwise return
                                    to caller.
$C972 (51570) [ESCTAB] \P17\        Table of ESCape codes
$C983 (51587) [ESCCHAR] \P17\       Table of corresponding control codes-high bit set for "remain in ESCape mode"
$C994 (51604) [PSTATUS] \SE\        pascal check if ready for input or output, return 3 in X
                                    if not ready (ILLEGAL OPERATION)
$C9A6 (51622) [PHOOK] \SE\          Pascal 1.0 output hook
$C9B7 (51639) [NOESC] \SE\          Monitor routine to process normal characters. Checks for copy char (right arrow),
                                    literal input, double quotes to turn literal input off/on, and restricted case
                                    input before storing in CHAR and returning to caller
$C9DF (51679) [B.CHKCAN] \L\        Monitor routine to check for cancelling literal mode
$C9F7 (51703) [B.FLIP] \L\          Monitor routine to switch the literal mode
$CA02 (51714) [B.CANLIT] \L\        Monitor routine to cancel literal mode
$CA0A (51722) [B.FIXCHAR] \L\       Monitor routine to up/shift the character in non-literal or restrict mode
$CA24 (51748) [B.INRET] \L\         Monitor routine to return to caller from input
$CA27 (51751) [GETPRIOR] \SE\       Monitor S/R to get the character before the cursor. Uses OURCH, OURCV; destroys A,
                                    TEMP1; outputs BEQ if character is double quote, BNE if not. Used for changing
                                    literal mode if backspacing over a double quote.
$CA4A (51786) [PINIT1.0] \SE\       Pascal initialization 1.0
$CA4F (51791) [PINIT] \SE\          Pascal initialization 1.1
$CA51 (51793) [PINIT2] \L\          Set up for running Pascal, set mode, set window, zero page, check for card,
                                    return X=9 (NO DEVICE) if missing, turn on card, set normal lower case mode,
                                    home and clear screen, put cursor on screen and return.
$CA74 (51828) [PREAD] \SE\          Pascal input-Get a character, remove high bit, store in CHAR, if 1.1 return
                                    "$C3" in X, 1.0 return CHAR in A
$CA8E (51854) [PWRITE] \SE\         Pascal output-Set zero page, turn cursor off, check GOTOXY Mode and process if
                                    necessary, check if GOTOXY and start if true, else store it on screen, increment
                                    cursor horizontal, check if transparent mode and do carriage return/line feed if
                                    necessary, replace the cursor and return.
------------------------------------------------------------------------------------------------
```

```
HEX LOCN (DEC LOCN) [NAME]  \USE-TYPE\ - DESCRIPTION
------------------------------------------------------------------------------------------

$CB15 (51989) [GETKEY] \SE\        Monitor S/R to read the keyboard, incrementing the random locations while waiting,
                                   load the char into A, clear the keyboard strobe and return
$CB24 (52004) [TESTCARD] \SE\      Monitor S/R to test for presence of 80 column card, destroys A,Y; returns BEQ if
                                   card is there, BNE if not.
$CB51 (52049) [BASCALC] \SE\       Monitor S/R to calculate base address for screen line using OURCV.
                                   Stores result in BASL/BASH.
$CB54 (52052) [BASCALCZ] \SE\      Monitor S/R to calculate base address for screen line using CV. Checks for 40/80
                                   column mode and if IRQ is enabled and not in Pascal, uses SNIFFIRQ to check
                                   for interrupts.
$CB99 (52121) [CTLCHAR] \SE\       Monitor S/R to process command control characters. Char in A to process,
                                   returns BCC if executed, BCS if not control command.
$CBB6 (52150) [CTLXFER] \L\        Monitor routine to push CTLADH and CTLADL onto stack for control routine address
                                   and execute a fake RTS.
$CBBC (52156) [X.BELL] \SE\        Monitor S/R to beep speaker, same as F8: BELL1
$CBCF (52175) [WAIT] \SE\          Monitor S/R to wait depending on A. Same as F8: WAIT
$CBDB (52187) [X.BS] \SE\          Monitor S/R to execute a backspace
$CBEC (52204) [X.CR] \SE\          Monitor S/R to execute a carriage return
$CC0D (52237) [X.EM] \SE\          Monitor S/R to execute HOME
$CC1A (52250) [X.SUB] \SE\         Monitor S/R to execute clear line
$CC26 (52262) [X.FS] \SE\          Monitor S/R to execute a forward space
$CC34 (52276) [X.US] \SE\          Monitor S/R to execute a reverse linefeed
$CC49 (52297) [X.SO] \SE\          Monitor S/R to execute "normal video"
$CC52 (52306) [X.SI] \SE\          Monitor S/R to execute "inverse video"
$CC5F (52319) [CTLADL] \P24\       Table of low byte addresses for control characters subroutines: 0 = Invalid
$CC78 (52344) [CTLADH] \P24\       Table of high byte addresses for control character subroutines: 0 = Invalid
$CC91 (52369) [X.LF] \SE\          Monitor S/R to execute linefeed
$CCA4 (52388) [SCROLLUP] \SE\      Monitor S/R to scroll the screen up one line
$CCAA (52394) [SCROLLDN] \SE\      Monitor S/R to scroll the screen down one line
$CCAE (52398) [SCROLL1] \L\        Monitor routine to check for 40/80 columns
$CCB8 (52408) [SCROLL2] \L\        Monitor routine to scroll 40 columns
$CCC0 (52416) [SCROLL80] \L\       Monitor routine to scroll the other 40 columns
$CCD1 (52433) [SCRLSUB] \SE\       Monitor S/R to scroll only 40 column active window
$CD11 (52497) [X.SCRLRET] \L\      Monitor rotuine to clear top or bottom line (depending on scroll up or down)
                                   Return to user via BASCALC.
$CD23 (52515) [X.VT] \SE\          Monitor S/R to clear to end of page
$CD42 (52546) [X.FF] \SE\          Monitor S/R to home the cursor. Returns via X.VT to clear screen.
$CD48 (52552) [X.GS] \SE\          Monitor S/R to clear to end of line
$CD4E (52558) [X.GSEOLZ] \SE\      Monitor S/R to clear entire line
$CD59 (52569) [X.DC1] \SE\         Monitor S/R to set 40 column mode
$CD77 (52599) [X.DC2] \SE\         Monitor S/R to set 80 column mode
$CD90 (52624) [X.NAK] \SE\         Monitor S/R/ to quit 80 column card
$CD9B (52635) [FULL80] \SE\        Monitor S/R to set full 80 column window parameters
$CDAA (52650) [QUIT] \SE\          Monitor S/R to restore 40 column window,convert 80 to 40 if needed, set cursor at
                                   bottom left corner, reset video and keyboard to old mode
```

HEX LOCN (DEC LOCN) [NAME] \USE-TYPE\	- DESCRIPTION
$CDDB (52699) [SCRN84] \SE\	Monitor S/R to convert 80 column screen to 40 column screen. Moves leftmost 40 characters to TXTPAGE1
$CE0A (52746) [ATEFOR] \SE\	Monitor S/R to convert one line from 80 to 40 columns
$CE22 (52770) [GET84] \SE\	Monitor S/R to move one character from 80 window to 40 window
$CE32 (52786) [SCRN48] \SE\	Monitor S/R to convert 40 column screen to 80 column screen. Moves whole 40 character screen to left most 40 positions on 80 column screen
$CE63 (52835) [FORATE] \SE\	Monitor S/R/ to convert one line from 80 to 40 columns
$CE91 (52881) [CLRHALF] \SE\	Monitor S/R to clear right half of both screen pages
$CEA3 (52899) [D048] \L\	Monitor S/R to move one character from 80 to 40 columns
$CEAF (52911) [SETCH] \SE\	Monitor S/R to set OURCH and CH. In 40 column mode sets to A value. In 80 column mode, sets to 0 unless less than 8 from end of line, in which case moves up near right
$CEDD (52957) [INVERT] \SE\	Monitor S/R to invert the character at the current screen location: CH,CV
$CEF2 (52978) [STORCHAR] \SE\	Monitor S/R to store character in A at screen horizontal position Y.
$CF01 (52993) [PICK] \SE\	Monitor S/R to read the character at screen position Y = horizontal, returns with character in A
$CF06 (52998) [SCREENIT] \SE\	Monitor S/R/ to either store character on screen or read character from screen. V clear for pick, V set for store, character in A for store, Y = CH position. Saves Y and checks for mode. 40 branches to SCREEN40, 80 falls through to SCREEN80
$CF0E (53006) [SCREEN80] \L\	Monitor routine to calculate which page, and if V set, branch to STOR80, otherwise read the character from the screen and return.
$CF2A (53034) [STOR80] \L\	Monitor routine to store the character on the screen.
$CF37 (53047) [SCREEN40] \L\	Monitor routine to get cursor position, and if V set, branch to STOR40, otherwise read the character from the screen and return.
$CF4A (53066) [STOR40] \L\	Monitor routine to store the character on the screen.
$CF52 (53074) [ESCON] \SE\	Monitor S/R to save current character in CHAR and put inverse "+" on screen. Returns via ESCRET.
$CF65 (53093) [ESCOFF] \SE\	Monitor S/R to replace original character back on the screen that was saved in CHAR. Falls through to ESCRET.
$CF6E (53102) [ESCRET] \L\	Monitor routine to put character on screen and return.
$CF78 (53112) [COPYROM] \SE\	Monitor S/R to copy the F8 ROM to the language card. Destroys X and Y. Uses CSWL/CSWH (which it saves) as hook for transfer. Sets ROM/RAM banks for transfer, moves the bytes, and resets the language card to it's previous state before returning.
$CFC8 (53192) [PSETUP] \SE\	Monitor S/R to set up zero page for Pascal operation. Checks 40-80 columns, sets INVFLG, and updates BASL/BASH before returning.
$CFEA (53226) [F.TABLE] \P9\	Table of addresses for ESCape functions in 40 column mode. Entries at $CFF0-1 are used by SCROLL (Label = PLUSMINUS1).
$CFF3 (53235) [B.TABLE] \P9\	Table of addresses for ESCape functions in 80 column mode. Entries at $CFF9-A are used by SCROLL (Label = WNDTAB).

MICRO SURVEY: JUNE 1984

HELP YOURSELF! To keep MICRO in touch with the rapidly changing computer world so that we can give you the information you need, please take a few minutes to fill in this questionnaire and mail it back to us. THANK YOU for your time.

DEMOGRAPHICS

1. What is your age?
☐ –19 ☐ 20-29 ☐ 30-39 ☐ 40-49 ☐ 50-59 ☐ 60+

2. What is your occupation?
☐ Programmer/analyst ☐ Engineer ☐ Technician
☐ Professor/teacher ☐ Lawyer ☐ Doctor
☐ Business person ☐ Student ☐ Self Employed
☐ Other ____________________

3. What is your formal educational level?
☐ Fewer than 12 years ☐ High school graduate ☐ Associate degree ☐ Bachelor's degree ☐ Para-professional degree
☐ Advanced degree

4. What is your annual household income before taxes?
☐ Less than $20,000 ☐ $20,000-29,999 ☐ $30,000-39,999 ☐ $40,000-49,999 ☐ $50,000+

COMPUTER INFORMATION

5. What microcomputer(s) do you use?
☐ AIM ☐ Apple II ☐ Atari (Model) __________ ☐ Commodore 64 ☐ KIM ☐ Macintosh ☐ OSI (Model)N__________
☐ PET/CBM ☐ SYM ☐ VIC ☐ TRS-80 Color Computer ☐ Other 6502 __________ ☐ Other 6809 __________
☐ Other computers/processors ____________________

6. Where do you use the above computer(s)?
☐ Home ☐ Work ☐ School ☐ Other ____________________

7. Approximately how much have you spent on your computer hardware so far?
☐ –$500 ☐ $500-999 ☐ $1,000-1,999 ☐ $2,000-2,999 ☐ $3,000-3,999 ☐ $4,000-4,999 ☐ $5,000-9,999 ☐ $10,000+

8. Approximately how much do you expect to spend on your computer hardware in the next year?
☐ –$500 ☐ $500-999 ☐ $1,000-1,999 ☐ $2,000-2,999 ☐ $3,000-3,999 ☐ $4,000-4,999 ☐ $5,000-9,999 ☐ $10,000+

9. What additions have you made to your basic system?
☐ Disk Drives ☐ Modem ☐ Serial Interface ☐ Parallel Interface ☐ RAM cards ☐ 6809 card ☐ 68000 card ☐ Z80 card
☐ Hard Disk ☐ Graphics Tablet ☐ Printer (type) ____________________

10. What additional hardware changes or upgrades do you plan to make to your system?
☐ Disk Drives ☐ Modem ☐ Serial Interface ☐ Parallel Interface ☐ RAM cards ☐ 6809 card ☐ 68000 card ☐ Z80 card
☐ Hard Disk ☐ Graphics Tablet ☐ Printer (type) ____________________
☐ Other hardware ____________________

11. Have you ever constructed a computer, computer board, or major computer equipment? ☐ Yes ☐ No
If yes, describe ____________________

12. Have you switched from one computer to another? ☐ Yes ☐ No
If yes, explain ____________________

13. Approximately how much have you spent on your computer software so far?
☐ –$200 ☐ $200-499 ☐ $500-999 ☐ $1,000-1,999 ☐ $2,000+

14. Approximately how much do you expect to spend on computer software in the next year?
☐ –$200 ☐ $200-499 ☐ $500-999 ☐ $1,000-1,999 ☐ $2,000+

15. How do you use your computer equipment?
☐ Business ☐ Software Development ☐ Hardware Development ☐ Telecommunications ☐ Entertainment ☐ Education
☐ Hobby ☐ Graphics ☐ Word Processing ☐ Database Management ☐ Other ____________________

16. What languages do you use?
☐ BASIC ☐ Pascal ☐ Forth ☐ C ☐ COBOL ☐ APL ☐ LOGO ☐ LISP ☐ Fortran
☐ 6502 Assembler ☐ 6809 Assembler ☐ 68000 Assembler ☐ Other ____________________

17. In an average week, about how many hours do you spend on a microcomputer performing the following operations?

	0-2	2-4	4-8	8-10	More
Programming for fun or self-education	☐	☐	☐	☐	☐
Programming professionally	☐	☐	☐	☐	☐
Using packaged programs in business	☐	☐	☐	☐	☐
Using packaged programs at home	☐	☐	☐	☐	☐
Using packaged programs for education	☐	☐	☐	☐	☐
Playing games	☐	☐	☐	☐	☐
Other	☐	☐	☐	☐	☐

18. If you write programs, what type of programming do you spend most of your time developing?
☐ Business applications ☐ Games ☐ Software development utilities ☐ Other ____________________

19. In an average month how much time do you spend with MICRO?
☐ Less than 2 hours ☐ 2-4 hours ☐ 4-8 hours ☐ More than 8 hours

20. How would you rate your present microcomputer knowledge?
Software: ☐ Elementary ☐ Intermediate ☐ Advanced
Hardware: ☐ Elementary ☐ Intermediate ☐ Advanced

Magazine Information

21. How long have you subscribed to or read MICRO?
☐ Less than 6 months ☐ 6 months to 1 year ☐ Over 1 year ☐ Over 2 years ☐ Over 3 years ☐ From the beginning

22. How did you get your current issue?
☐ Subscription ☐ Computer store ☐ Newsstand ☐ Bookstore ☐ Borrowed ☐ Library

23 To what other computer publications do you subscribe?
☐ BYTE ☐ Commander ☐ Compute! ☐ Creative Computing ☐ Dr. Dobbs ☐ In'Cider
☐ Kilobaud Microcomputing ☐ Nibble ☐ Personal Computing ☐ Popular Computing ☐ RUN ☐ Softalk
☐ 80 Micro ☐ 68 Micro ☐ Other(s) ____________________

24. Please rate the following parts of MICRO as to their interest, with 5 being very interesting and 1 not at all interesting.

	5	4	3	2	1		5	4	3	2	1		5	4	3	2	1
News	☐	☐	☐	☐	☐	Advertisements	☐	☐	☐	☐	☐	Hardware & Software Catalogs	☐	☐	☐	☐	☐
Articles	☐	☐	☐	☐	☐	Columns	☐	☐	☐	☐	☐	System specific information	☐	☐	☐	☐	☐
Reviews	☐	☐	☐	☐	☐	Editorials	☐	☐	☐	☐	☐	New Publications	☐	☐	☐	☐	☐

25. Please rate the following kinds of articles as to their interest, with 5 being very interesting and 1 not at all interesting.

	5	4	3	2	1		5	4	3	2	1		5	4	3	2	1
Hardware tutorials	☐	☐	☐	☐	☐	Applications	☐	☐	☐	☐	☐	Review articles	☐	☐	☐	☐	☐
Programming techniques	☐	☐	☐	☐	☐	Games	☐	☐	☐	☐	☐	Programs	☐	☐	☐	☐	☐
Utilities	☐	☐	☐	☐	☐	News & Information	☐	☐	☐	☐	☐	Languages	☐	☐	☐	☐	☐

26. Is MICRO ☐ too technical ☐ not technical enough ☐ just right?

27. What new areas would you like to have MICRO cover: ______________________________

____

28. Do you key in the longer programs published in MICRO? ☐ Yes ☐ No

29. Would you be willing to pay extra to receive MICRO's programs in diskette form? ☐ Yes ☐No

30. Overall, how do you feel about MICRO? How useful is MICRO to you?

____

____

____

____

Fold Here

31. What are your favorite software packages in the following categories:

Software Package	First Choice	Second Choice	Third Choice
Data Base Manager			
Word Processors			
Editor/Assembler			
Spread Sheet			
Monitor/Debugger			
Communications			
Other ____________			

For the ______________________ computer.

Please feel free to write comments, suggestions and so forth on an additional page of paper and attach it to this form.

Fold Here

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD

FIRST CLASS PERMIT NO. 60, CHELMSFORD, MA

POSTAGE WILL BE PAID BY ADDRESSEE

MICRO™
P.O. Box 6502
Chelmsford, MA 01824

Tape Here

Changes in the F800 ROM

```
$F7FF (63487) [?]                        was $D7, is now $78, appears to be unused
$FA75-$FA7A (64117-64122) [RESET]        A change in the RESET code to allow for the presence of an 80 column card. Does a
                                         JSR to GOTOCX Y=5
$FB0A-$FB0D (64226-64269) [TITLE]        APPLE ][ -> Apple ][
$FB51-$FB54 (64337-64340) [SETWND]       A change in the SETWND code to allow for the presence of an 80 column card. Does
                                         a branch to GOTOCX Y=8
$FBA3 (64419) [ESCNOW]                   A change in the ESCNOW code to allow for i,j,k,m and arrow keys. Does JSR to RSDEC
                                         which is the old KEYIN2
$FBB3 (64435) [VERSION]                  ID code for check on which kind of Apple it is //e=$06 ][+=$EA ][=$38
$FBB4-$FBC0 (64436-64448) [GOTOCX]       Formerly NOPs, now code to save current ROM states, set interrupts, turn on CX00
                                         ROMS and JMP to C100:new code for 80 cols. Requires function code to be in Y Reg.
$FC42-$FC45 (64578-64581) [CLREOP]       Changed to branch to GOTOCX Y=0
$FC46-$FC57 (64582-64599) [COPYRT]       Notice of copyright "(C) 1981-82, APPLE"
$FC58-$FC5B (64600-64603) [HOME]         Changed to branch to GOTOCX Y=1
$FC5C-$FC61 (64604-64609) [AUTHOR1]      "RICK A" for Rick Auricchio
$FC70-$FC71 (64624-64625) [SCROLL]       Changed to jump to GOTOCX Y=2
$FC72-$FC74 (64626-64628) [XGOTOCX]      A JMP to GOTOCX for long branching purposes
$FC75-$FC9B (64629-64667) [SNIFFIRQ]     IRQ Sniffer for Video Code: A new routine to check the current video mode,
                                         CXROM usage, and check for interrupts
$FC9C-$FC9D (64668-64669) [CLREOL]       Changed to branch to GOTOCX Y=3
$FC9E-$FCA7 (64670-64679) [CLREOLZ]      Changed to branch to GOTOCX Y=4
$FD1B-$FD20 (64795-64800) [KEYIN]        Changed to jump to GOTOCX Y=6 KEYIN no longer falls through to KEYIN2.
$FD21-$FD28 (64801-64808) [RDESC]        Formerly KEYIN2, changed to jump to GOTOCX Y=7
$FD29-$FD2D (64809-64813) [FUNCEXIT]     Return from GOTOCX here: A new routine that restores the CXROM bank and the
                                         IRQ before an RTS to the calling routine.
$FD30 (64816) [ESC]                      A change to JSR to RDESC instead of RDKEY
$FD42-$FD43 (64834-64835) [NOTCR]        A change to NOPs of the cursor inverse mode. No longer needed now that the
                                         cursor is a standard character.
$FD83 (64899) [CAPTST] \P1\              A change in the input AND mask that used to convert lower case input to upper case
$FEAF (65199) [CKSUMFIX] \P1\            Correct CKSUM at create time.
$FEC5-$FEC9 (65221-65225) [AUTHOR2]      "Bryan" for Bryan Stearns.
```

```
NAME (DEC LOCN) [HEX LOCN] \USE-TYPE\ - DESCRIPTION
------------------------------------------------------------------------------------------

? (63487) [$F7FF]                         was $D7, is now $78, appears to be unused
ATEFOR (52746) [$CE0A] \SE\               Monitor S/R to convert one line from 80 to 40 columns
AUTHOR1 (64604-64609) [$FC5C-$FC61]       "RICK A" for Rick Auricchio
AUTHOR2 (65221-65225) [$FEC5-$FEC9]       "Bryan" for Bryan Stearns.
B.CANLIT (51714) [$CA02] \L\              Monitor routine to cancel literal mode
B.CHKCAN (51679) [$C9DF] \L\              Monitor routine to check for cancelling literal mode
B.CLREOL (49619) [$C1D3] \SE\             Entry point for monitor routine to clear to end of line
B.CLEOLZ (49625) [$C1D9] \SE\             Entry point for monitor routine to clear entire line
B.CLREOP (49633) [$C1E1] \SE\             Entry point for monitor routine to clear to end of page
B.ESCFIX (49774) [$C26E] \SE\             Monitor S/R to map i,j,k,m and <-,^,->, and V into I,J,K,M for cursor movement
B.INPUT (51461) [$C905] \SE\              Monitor routine to inverse char at current position, get a char from the keyboard,
                                          remove cursor, and process char, including ESCapes. If not ESC then JMP to NOESC.
B.FIXCHAR (51722) [$CA0A] \L\             Monitor routine to up/shift the character in non-literal or restrict mode
B.FLIP (51703) [$C9F7] \L\                Monitor routine to switch the literal mode
B.FUNC (49408) [$C100] \SE\               Entry point for all routines with code in Y.Check first for KEYIN Y=6
B.FUNC1 (49681) [$C211] \SE\              Pushes $C1 on stack, and low byte address of the function -1 by looking up in
                                          B.TABLE indexed by Y. Then does fake RTS to routine.
B.FUNCNE (49422) [$C10E] \SE\             Test for card. If present, use the new routines, if not, old routines
B.FUNCNK (49415) [$C107] \SE\             Check for ESCape-fix Y=7
B.FUNCO (49572) [$C1A4] \SE\              Entry point to new routines. Sets the IRQ mode and screen holes, Y reg.
B.GETCH (49678) [$C20E] \SE\              Save CH in screenhole
B.INRET (51748) [$CA24] \L\               Monitor routine to return to caller from input
B.KEYIN (49800) [$C288] \SE\              Monitor routine to read a key with new additions to save CX bank status, check
                                          interrupt status, put new cursor ASC"$FF" on screen, JSR to KEYDLY (old RDKEY)
B.OLDFUNC (49439) [$C11F] \SE\            Pushes $C1 on stack, and low byte address of the function -1 by looking up in
                                          F.TABLE indexed by Y. Then does fake RTS to routine.
B.RESETX (49716) [$C234] \SE\             Monitor routine to reset system, checks for "Apple" keys for cold start, else does
                                          warm restart without diagnostics, blasts memory from BFXX down to stack, checks
                                          80 col board to see if CX ROM needs resetting, and returns
B.SCROLL (49613) [$C1CD] \SE\             Entry point for monitor routine to scroll up one line
B.SETWND (49639) [$C1E7] \SE\             Entry point for monitor routine to set text window
B.SETWNDX (49689) [$C219] \SE\            Monitor S/R to set normal text window 40/80 columns
B.TABLE (53235) [$CFF3] \P9\              Table of addresses for ESCape functions in 80 column mode. Entries at $CFF9-A are
                                          used by SCROLL (Label = WNDTAB).
B.VECTOR (49663) [$C1FF] \SE\             Monitor S/R to check on 80 col use and get current Cursor Horizontal position (CH)
BASCALC (52049) [$CB51] \SE\              Monitor S/R to calculate base address for screen line using OURCV.
                                          Stores result in BASL/BASH.
BASCALCZ (52052) [$CB54] \SE\             Monitor S/R to calculate base address for screen line using CV. Checks for 40/80
                                          column mode and if IRQ is enabled and not in Pascal, uses SNIFFIRQ to check
                                          for interrupts.
BASICENT (49943) [$C317] \SE\             BASIC I/O entry point, saves CHAR, A, Y, X, and P, pulls P from stack, checks IRQ
                                          status, and sets appropriately.
BASICENT2 (49974) [$C336] \SE\            Turns off any slots using C8 area, sets C8SLOT to $C3, checks INIT flag, and jumps
                                          to warm or cold BASIC in C8 ROM

------------------------------------------------------------------------------------------
AUTHOR1 - BASICENT2               Prof. Luebbert's "What's Where in the Apple" //e        ALPHABETICAL GAZETTEER
```

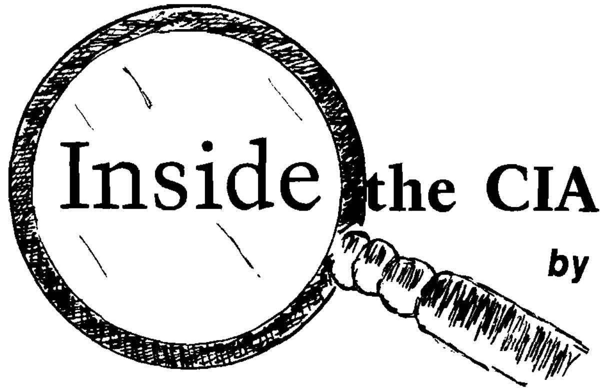
```
NAME (DEC LOCN) [HEX LOCN] \USE-TYPE\ - DESCRIPTION
------------------------------------------------------------------------------------------------

DIAGS (49761) [$C261] \SE\          Entry point for monitor S/R diagnostics
DO48 (52899) [$CEA3] \L\            Monitor routine to move one character from 80 to 40 columns
ESC (64816) [$FD30]                 A change to JSR to RDESC instead of RDKEY
ESCAPING (51480) [$C918] \SE\       Monitor routine to process ESCape command sequences. Places ESCape
                                    cursor on screen, GETs a command key, puts lower case into upper,
                                    checks the ESCTAB for a valid character. If the char is there, load A with the Y
                                    index into ESCCHAR, and "print" the control character, if its not, check for "T",
                                    "R" and "CTRL-Q" special functions and process, if its not, return to caller.
                                    If the ESCCHAR entry has the high bit set, return to ECSAPING, otherwise return
                                    to caller.
ESCCHAR (51587) [$C983] \P17\       Table of corresponding control codes-high bit set for "remain in ESCape mode"
ESCIN (49792) [$C280] \P4\          Table of arrow keys
ESCNOW (64419) [$FBA3]              A change in the ESCNOW code to allow for i,j,k,m and arrow keys. Does JSR to RSDEC
                                    which is the old KEYIN2
ESCOFF (53093) [$CF65] \SE\         Monitor S/R to replace original character back on the screen that was saved
                                    in CHAR. Falls through to ESCRET.
ESCON (53074) [$CF52] \SE\          Monitor S/R to save current character in CHAR and put inverse "+" on screen.
                                    Returns via ESCRET.
ESCOUT (49796) [$C284] \P4\         "J,K,M,I" translations for arrows
ESCRET (53102) [$CF6E] \L\          Monitor routine to put character on screen and return.
ESCTAB (51570) [$C972] \P17\        Table of ESCape codes
F.CLREOL (49533) [$C17D] \SE\       Monitor S/R to clear to end of line.
F.CLEOLZ (49564) [$C19C] \SE\       Monitor S/R to clear entire line.
F.CLREOP (49449) [$C129] \SE\       Monitor S/R to clear from the cursor to the end of page.
F.GORET (49569) [$C1A1] \L\         Exit routine to F.RETURN
F.HOME (49475) [$C143] \SE\         Clear scroll window to blanks. Set cursor to top left corner.
F.RETURN (49899) [$C2EB] \SE\       Monitor routine to exit from CX ROM routines either leaving I/O disabled or
                                    enabling it if it was on entry
F.SCROLL (49485) [$C14D] \SE\       Monitor S/R to scroll up one line.
F.SETWND (49546) [$C18A] \SE\       Monitor S/R to set normal low-resolution graphics window, cursor bottom left.
F.TABLE (53226) [$CFEA] \P9\        Table of addresses for ESCape functions in 40 column mode. Entries at $CFF0-1 are
                                    used by SCROLL (Label = PLUSMINUS1).
FORATE (52835) [$CE63] \SE\         Monitor S/R/ to convert one line from 80 to 40 columns
FULL80 (52635) [$CD9B] \SE\         Monitor S/R to set full 80 column window parameters
FUNCEXIT (64809-64813) [$FD29-$FD2D] Return from GOTOCX here: A new routine that restores the CXROM bank and the
                                    IRQ before an RTS to the calling routine.
GET84 (52770) [$CE22] \SE\          Monitor S/R to move one character from 80 window to 40 window
GETKEY (51989) [$CB15] \SE\         Monitor S/R to read the keyboard, incrementing the random locations while waiting,
                                    load the char into A, clear the keyboard strobe and return
GETPRIOR (51751) [$CA27] \SE\       Monitor S/R to get the character before the cursor. Uses OURCH, OURCV; destroys A,
                                    TEMP1; outputs BEQ if character is double quote, BNE if not. Used for changing
                                    literal mode if backspacing over a double quote.

------------------------------------------------------------------------------------------------
```

NAME (DEC LOCN) [HEX LOCN] \USE-TYPE\	DESCRIPTION
GOTOCX (64436-64448) [$FBB4-$FBC0]	Formerly NOPs, now code to save current ROM states, set interrupts, turn on CX00 ROMS and JMP to C100:new code for 80 cols. Requires function code to be in Y Reg.
HOME (64600-64603) [$FC58-$FC5B]	Changed to branch to GOTOCX Y=1
INVERT (52957) [$CEDD] \SE\	Monitor S/R to invert the character at the current screen location: CH,CV
JPINIT (49995) [$C34B] \SE\	Pascal INIT
JPREAD (50001) [$C351] \SE\	Pascal READ
JPSTAT (50013) [$C35D] \SE\	Pascal STATUS
JPWRITE (50007) [$C357] \SE\	Pascal WRITE
KEYDLY (49862) [$C2C6] \SE\	Monitor routine to get a key from KBD, also checking interrupts, and still incrementing RNDL and RNDH, the random locations
KEYIN (64795-64800) [$FD1B-$FD20]	Changed to jump to GOTOCX Y=6 KEYIN no longer falls through to KEYIN2.
MODE (1275) [$4FB] \P1\	Current operating mode acording bits set.
MOVE (50019) [$C363] \SE\	Monitor S/R to move memory across memory banks. Call with A1 = Source start, A2 = Source end, A4 = Destination start, Carry set for Main to Card, Carry clear for Card to Main.
NOESC (51639) [$C9B7] \SE\	Monitor routine to process normal characters. Checks for copy char (right arrow), literal input, double quotes to turn literal input off/on, and restricted case input before storing in CHAR and returning to caller
NOTCR (64834-64835) [$FD42-$FD43]	A change to NOPs of the cursor inverse mode. No longer needed now that the cursor is a standard character.
OLDBASH (2043) [$7FB] \P1\	Pascal saved BASH
OLDBASL (1915) [$77B] \P1\	Pascal saved BASL
OLDCH (1147) [$47B] \P1\	Old CH set for user
128KJMP (49937) [$C311] \P6\	Jump table for 128K support routines
OURCH (1403) [$57B] \P1\	80 column CH
OURCV (1531) [$5FB] \P1\	Cursor vertical
PASFPT (49931) [$C30B] \P6\	Pascal 1.1 firmware protocol table
PHOOK (51622) [$C9A6] \SE\	Pascal 1.0 output hook
PICK (52993) [$CF01] \SE\	Monitor S/R to read the character at screen position Y = horizontal, returns with character in A
PINIT (51791) [$CA4F] \SE\	Pascal initialization 1.1
PINIT1 (51200) [$C800] \SE\	Pascal 1.0 init
PINIT1.0 (51786) [$CA4A] \SE\	Pascal initialization 1.0
PINIT2 (51793) [$CA51] \L\	Set up for running Pascal, set mode, set window, zero page, check for card, return X=9 (NO DEVICE) if missing, turn on card, set normal lower case mode, home and clear screen, put cursor on screen and return.
PJUMPS (49995-50018) [$C34B-$C362]	Pascal jump table
PREAD (51828) [$CA74] \SE\	Pascal input-Get a character, remove high bit, store in CHAR, if 1.1 return "$C3" in X, 1.0 return CHAR in A
PREAD1.0 (51272) [$C84B]	Pascal 1.0 input hook
PSETUP (53192) [$CFC8] \SE\	Monitor S/R to set up zero page for Pascal operation. Checks 40-80 columns, sets INVFLG, and updates BASL/BASH before returning.
PSTATUS (51604) [$C994] \SE\	pascal check if ready for input or output, return 3 in X if not ready (ILLEGAL OPERATION)

NAME (DEC LOCN) [HEX LOCN] \USE-TYPE\	DESCRIPTION
BASICINIT (51203) [$C803] \SE\	Checks the F8 ROM version, if not //e, copies ROM to RAM Card, and checks again, if still not good, hangs the system.
BASICINT (49920) [$C300] \SE\	Sets INIT Flag (V) and branches to BASIC I/O entry point
BASICOUT (49927) [$C307] \SE\	Clears INIT Flag (V) and branches to BASIC I/O entry point
BINIT1 (51222) [$C816] \SE\	Set up BASIC I/O in CSW and KSW to point to BASICENT in the C3 ROM and set text or graphics windows
BINIT2 (51280) [$C850] \L\	Check for 80 column mode and enable, if true
BINPUT (51446) [$C8F6] \SE\	Monitor routine to set MODE to BASIC input, get the cursor position, and CHAR
BIORET (51426) [$C8E2] \L\	Monitor routine to store cursor position, restore X, Y, and A and return to BASIC
BOUT (51350) [$C896] \SE\	Monitor S/R to set MODE to BASIC printing, falls through to BPRINT
BPNCTL (51404) [$C8CC] \SE\	Monitor S/R to reload CHAR (to get 8th bit, and print the char on the screen, Increments cursor horizontal and scrolls, if necessary
BPRINT (51361) [$C8A1] \SE\	Monitor S/R to output character in CHAR, checks for CTRL-S, clears high bit, checks for CTRL chars, if it is, process and return, if not, fall through to BPNCTL.
C8B2 (51316) [$C874] \L\	Monitor routine to check current CH and store it if different from OLDCH
C8B3 (51326) [$C87E] \L\	Monitor routine to check RAM card for correct version and, if not, recopy the F8ROM to RAM card , check again and hang if not correct.
C8B4 (51344) [$C890] \L\	Monitor routine to check carry, on clear-print a character, set-input a character
C8BASIC (51302) [$C866] \L\	Monitor routine to check mode and set 80 column store in case Integer BASIC cleared Also rounds WNDWDTH to next lower even, if odd in 80 column mode.
CAPTST (64899) [$FD83] \P1\	A change in the input AND mask that used to convert lower case input to upper case
CHAR (1659) [$67B] \P1\	In/Out character
CKSUMFIX (65199) [$FEAF] \P1\	Correct CKSUM at create time.
CLEARIT (51293) [$C85D] \L\	Monitor routine to set lower case mode, clear screen and clears carry
CLR80COL (49152) [$C000] \H1\	Disable 80 column store
CLR80VID (49164) [$C00C] \H1\	Disable 80 column video
CLRALTCHAR (49166) [$C00E] \H1\	Normal lower case, flash upper case
CLREOL (64668-64669) [$FC9C-$FC9D]	Changed to branch to GOTOCX Y=3
CLREOLZ (64670-64679) [$FC9E-$FCA7]	Changed to branch to GOTOCX Y=4
CLREOP (64578-64581) [$FC42-$FC45]	Changed to branch to GOTOCX Y=0
CLRHALF (52881) [$CE91] \SE\	Monitor S/R to clear right half of both screen pages
COPYROM (53112) [$CF78] \SE\	Monitor S/R to copy the F8 ROM to the language card. Destroys X and Y. Uses CSWL/CSWH (which it saves) as hook for transfer. Sets ROM/RAM banks for transfer, moves the bytes, and resets the language card to it's previous state before returning.
COPYRT (64582-64599) [$FC46-$FC57]	Notice of copyright "(C) 1981-82, APPLE"
CTLADH (52344) [$CC78] \P24\	Table of high byte addresses for control character subroutines: 0 = Invalid
CTLADL (52319) [$CC5F] \P24\	Table of low byte addresses for control characters subroutines: 0 = Invalid
CTLCHAR (52121) [$CB99] \SE\	Monitor S/R to process command control characters. Char in A to process, returns BCC if executed, BCS if not control command.
CTLXFER (52150) [$CBB6] \L\	Monitor routine to push CTLADH and CTLADL onto stack for control routine address and execute a fake RTS.
CX00ROM (49408-53247) [$C100-$CFFF]	\SB\ A new set of subroutines to handle the 80 column card and auxiliary memory in slot 3. It is entered from the GOTOCX subroutine in the F800 ROM which sets interrupts, turns on the CX00 ROMs, and JMPs to C100. Function code is in Y reg. Note: "B." routines are the new way. "F." routines are the old way. Stack has status of bank and IRQ. Uses A,Y registers.

NAME (DEC LOCN) [HEX LOCN] \USE-TYPE\	DESCRIPTION
PWRITE (51854) [$CA8E] \SE\	Pascal output-Set zero page, turn cursor off, check GOTOXY Mode and process if necessary, check if GOTOXY and start if true, else store it on screen, increment cursor horizontal, check if transparent mode and do carriage return/line feed if necessary, replace the cursor and return.
QUIT (52650) [$CDAA] \SE\	Monitor S/R to restore 40 column window,convert 80 to 40 if needed, set cursor at bottom left corner, reset video and keyboard to old mode
RD80COL (49176) [$C018] \H1\	Reads SET80COL
RD80VID (49183) [$C01F] \H1\	Reads SET80VID
RDCARDRAM (49155) [$C003] \H1\	Read RAM on card
RDESC (64801-64808) [$FD21-$FD28]	Formerly KEYIN2, changed to jump to GOTOCX Y=7
RDLCBNK2 (49169) [$C011] \H1\	Reads language card bank 2
RDLCRAM (49170) [$C012] \H1\	Reads language card RAM enable
RDMAINRAM (49154) [$C002] \H1\	Read RAM on mainboard
RDPAGE2 (49180) [$C01C] \H1\	Reads page 1/2 status
RDRAMRD (49171) [$C013] \H1\	Reads RAMREAD state
RDRAMWRT (49172) [$C014] \H1\	Reads BANKWRT state
RDTEXT (49178) [$C01A] \H1\	Reads Text mode
RDVBLBAR (49177) [$C019] \H1\	Reads VBL signal
RESET (64117-64122) [$FA75-$FA7A]	A change in the RESET code to allow for the presence of an 80 column card. Does a JSR to GOTOCX Y=5
SCREEN40 (53047) [$CF37] \L\	Monitor routine to get cursor position, and if V set, branch to STOR40, otherwise read the character from the screen and return.
SCREEN80 (53006) [$CF0E] \L\	Monitor routine to calculate which page, and if V set, branch to STOR80, otherwise read the character from the screen and return.
SCREENIT (52998) [$CF06] \SE\	Monitor S/R/ to either store character on screen or read character from screen. V clear for pick, V set for store, character in A for store, Y = CH position. Saves Y and checks for mode. 40 branches to SCREEN40, 80 falls through to SCREEN80
SCRLSUB (52433) [$CCD1] \SE\	Monitor S/R to scroll only 40 column active window
SCRN48 (52786) [$CE32] \SE\	Monitor S/R to convert 40 column screen to 80 column screen. Moves whole 40 character screen to left most 40 positions on 80 column screen
SCRN84 (52699) [$CDDB] \SE\	Monitor S/R to convert 80 column screen to 40 column screen. Moves leftmost 40 characters to TXTPAGE1
SCROLL (64624-64625) [$FC70-$FC71]	Changed to jump to GOTOCX Y=2
SCROLL1 (52398) [$CCAE] \L\	Monitor routine to check for 40/80 columns
SCROLL2 (52408) [$CCB8] \L\	Monitor routine to scroll 40 columns
SCROLL80 (52416) [$CCC0] \L\	Monitor routine to scroll the other 40 columns
SCROLLDN (52394) [$CCAA] \SE\	Monitor S/R to scroll the screen down one line
SCROLLUP (52388) [$CCA4] \SE\	Monitor S/R to scroll the screen up one line
SET80COL (49153) [$C001] \H1\	Enable 80 column store
SET80VID (49165) [$C00D] \H1\	Enable 80 column video
SETALTCHAR (49167) [$C00F] \H1\	Normal/inverse lower case, no flash
SETALTZP (49161) [$C009] \H1\	Set alternate zero page/stack
SETC8 (50155) [$C3EB] \SE\	Setup IRQ C800 protocol. Stores $C3 in C8SLOT.
SETCH (52911) [$CEAF] \SE\	Monitor S/R to set OURCH and CH. In 40 column mode sets to A value. In 80 column mode, sets to 0 unless less than 8 from end of line, in which case moves up near right

SETINTCXROM (49159) [$C007] \H1\	Set internal CX00 ROM
SETSLOTC3ROM (49163) [$C00B] \H1\	Enable C300 slot ROM
SETSTDZP (49160) [$C008] \H1\	Set standard zero page/stack
SETWND (64337-64340) [$FB51-$FB54]	A change in the SETWND code to allow for the presence of an 80 column card. Does a branch to GOTOCX Y=8
SNIFFIRQ (64629-64667) [$FC75-$FC9B]	IRQ Sniffer for Video Code: A new routine to check the current video mode1 CXROM usage and interrupt status
STOR40 (53066) [$CF4A] \L\	Monitor routine to store the character on the screen.
STOR80 (53034) [$CF2A] \L\	Monitor routine to store the character on the screen.
STORCHAR (52978) [$CEF2] \SE\	Monitor S/R to store character in A at screen horizontal position Y.
TEMP1 (1144) [$478] \P1\	A temporary storage location
TESTCARD (52004) [$CB24] \SE\	Monitor S/R to test for presence of 80 column card, destroys A,Y; returns BEQ if card is there, BNE if not.
TITLE (64226-64269) [$FB0A-$FB0D]	APPLE -> Apple
VERSION (64435) [$FBB3]	ID code for check on which kind of Apple it is //e=$06 +=$EA =$38
WAIT (52175) [$CBCF] \SE\	Monitor S/R to wait depending on A. Same as F8: WAIT
WRCARDRAM (49157) [$C005] \H1\	Write RAM on card
WRMAINRAM (49156) [$C004] \H1\	Write RAM on mainboard
X.BELL (52156) [$CBBC] \SE\	Monitor S/R to beep speaker, same as F8: BELL1
X.BS (52187) [$CBDB] \SE\	Monitor S/R to execute a backspace
X.CR (52204) [$CBEC] \SE\	Monitor S/R to execute a carriage return
X.DC1 (52569) [$CD59] \SE\	Monitor S/R to set 40 column mode
X.DC2 (52599) [$CD77] \SE\	Monitor S/R to set 80 column mode
X.EM (52237) [$CC0D] \SE\	Monitor S/R to execute HOME
X.FF (52546) [$CD42] \SE\	Monitor S/R to home the cursor. Returns via X.VT to clear screen.
X.FS (52262) [$CC26] \SE\	Monitor S/R to execute a forward space
X.GS (52552) [$CD48] \SE\	Monitor S/R to clear to end of line
X.GSEOLZ (52558) [$CD4E] \SE\	Monitor S/R to clear entire line
X.LF (52369) [$CC91] \SE\	Monitor S/R to execute linefeed
X.NAK (52624) [$CD90] \SE\	Monitor S/R/ to quit 80 column card
X.SCRLRET (52497) [$CD11] \L\	Monitor rotuine to clear top or bottom line (depending on scroll up or down) Return to user via BASCALC.
X.SI (52306) [$CC52] \SE\	Monitor S/R to execute "inverse video"
X.SO (52297) [$CC49] \SE\	Monitor S/R to execute "normal video"
X.SUB (52250) [$CC1A] \SE\	Monitor S/R to execute clear line
X.US (52276) [$CC34] \SE\	Monitor S/R to execute a reverse linefeed
X.VT (52515) [$CD23] \SE\	Monitor S/R to clear to end of page
XCOORD (1787) [$6FB] \P1\	X coordinate in GOTOXY routine
XFER (50096) [$C3B0] \SE\	Transfer program control from main board to card or vice versa. $3ED-$3EE is address to be executed upon transfer, carry set means transfer to card, carry clear means transfer to main board, V flag clear means use standard zero page/stack, V flag set means use alternate zero page/stack. Also uses $3ED-$3EE in destination bank. Enter via JMP not JSR.
XGOTOCX (64626-64628) [$FC72-$FC74]	A JMP to GOTOCX for long branching purposes
YSAV1 (31) [$1F] \P1\	Temporary storage for the Y register



Inside the CIA

by Ralph Tenny

Last month we examined several programming modes on the 6526 CIA used for I/O on the Commodore 64. The shift register (SR) on the CIA was examined briefly, in that we learned to toggle the SR output (SP) by setting the mode to input (SP high) or output (SP low). This toggle mode of operation is useful if you wish to output multiple bytes of parallel data on the eight Port B lines of U2 (User Port). That is, by using SP1 (U1), SP2 (U2), PA2 and PA3 (U2) as clock strobes, eight bit values can be latched in four different latches as shown in Figure 1. Simply program all the port lines (PB0-PB7) as outputs, then load the port with the output data. Pulse one of the four strobe lines and the data will be transferred to the strobed latch.

You should note a few things shown in Figure 1. First, all the 74LS374 latches are wired identically. Pin 11 is the Clock pin which causes the input data to be captured on the rising edge of the clock signal. Each clock line is driven from a different strobe output on the port, thus allowing different data to be saved in each latch. Pin 1 on each latch is the Output Enable line; these lines are all tied low to keep the latch permanently enabled (compare with Figure 2). Also note that output latches do not have to have tri-state outputs, but input latches (Figure 2) must be able to disconnect from the bus unless it is a dedicated input.

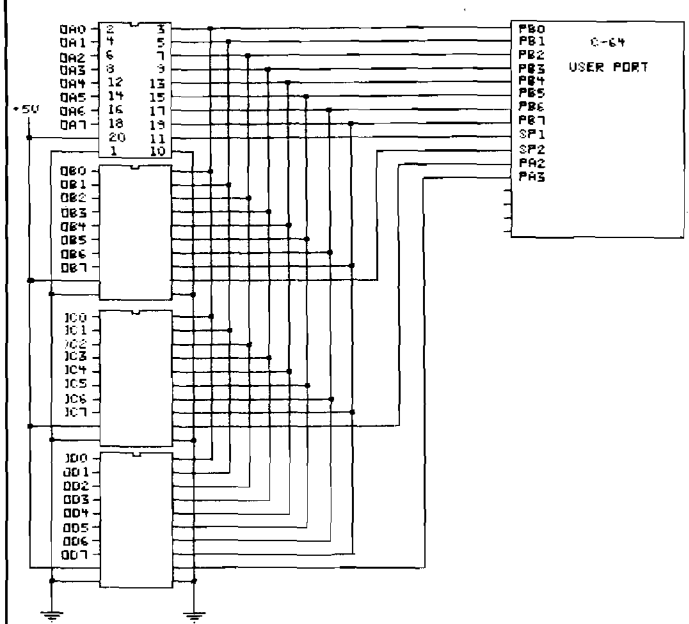


Figure 1. For 74LS374 latches selected by four unique strobes allows eight output lines to become the 32 output lines.

If you want to use the port for input instead of output, Figure 2 shows a tri-state latch connected for input. In this mode, the port must be programmed for input. Also, note that pin 1 (output enable) is separately controlled so the programmer can select one latch at a time to input from. The external hardware must load data into the latch by pulsing INLATCH, and it is wise to have some method of handshaking so the latch output doesn't change when the port tries to read the data. If four latches are attached, only one can be enabled at a time to avoid bus conflicts. If you use care in programming, it is possible to have (for example) two input and two output latches. This is what must happen: The input latches must be enabled *only* while they are being read, one at a time, and the port must be set for input. When outputting data, disable the output lines of the input latches, set the port for output, output the data, and clock the destination latch. If you want to go to extremes, one of the four output latches could be designated as a controller, enabling up to (for example) four inputs and four outputs, or any combination of eight 8-bit ports!

The CIA serial port is *not* a UART or ACIA, and requires some understanding to use. If you want to communicate with a computer, it is probably easier to

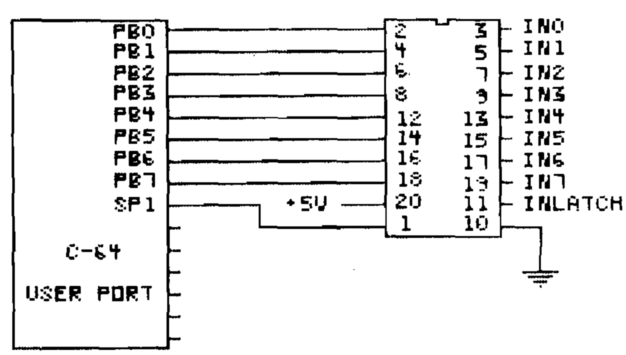


Figure 2. The 74LS374 latch can also be used for input if data is strobed in by external hardware and the output enable is controlled by the C-64 User Port.

write a software UART program which toggles a single line for output and receives on another line. The specific problems you will have using the SR for communication are:

1. The SR outputs 8-bit words only - normal asynchronous communication uses 10 bits minimum.

2. The SR inputs only 8 bits, so incoming asynchronous data will be scrambled. In addition, reliable asynchronous input involves an input clock 16 times the data rate; the CIA Shift Register is a synchronous device which depends upon the external source to furnish a properly timed input clock, one pulse per bit.

So, what is the SR good for? This port can be used as an I/O expander as shown in Figure 3. The output data from the SR becomes valid as the CNT line switches low, so the shift register used is positive-edge triggered and will accept data directly as shown. Also, the SR outputs the MSB (most significant bit) first, and assumes that incoming data has the same organization. If you use a negative-edge triggered shift register, the CNT line must be inverted. The shift registers used in Figure 3 are CD4015 CMOS parts, and are internally organized as two four-bit shift registers. Each successive section is cascaded with the previous one, and data is passed down the line. In order to output data to the circuit of Figure 3, the CIA SR is loaded with data for the output port B. This byte is shifted out; then, when the data for Port A is loaded, the Port B data is shifted into B and A's data is shifted into A. You need to realize that if the changing data will affect the external hardware, the scheme shown won't be acceptable. In that situation, the CD4094 is a four-bit register which allows data to be shifted in, then strobed onto the output lines when shifting is done. To use the SR for input, the external circuit must present data at the SR pin and then clock the CNT pin a short time later.

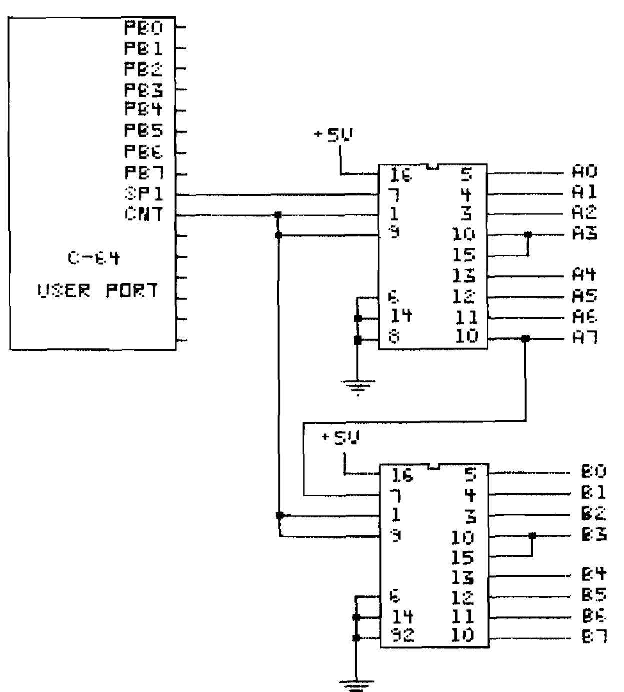


Figure 3. Subject to the limitations discussed in the text, the CIA ShiftRegister section can be used as an output expander.

A major advantage of the SR is that it can operate unattended in either polled or interrupt modes. Instead of using software timing loops to drive the serial port, the CIA uses Timer A in the free-running mode to drive the CNT pin and shift data out. After data has been transmitted or received, bit 3 of the Interrupt Control Register (ICR) is set high. If the CIA has been enabled for interrupt and the IRQ line is not masked, the processor will be interrupted. If you are outputting data, writing new data to the SR clears the interrupt bit and initiates the next transmission. Polled operation of the SR on input would require another CIA output line to be used for handshaking; otherwise it is possible to lose successive data bytes if the register isn't cleared in time.

This is an abbreviated step-by-step procedure for using the Shift Register in the non-interrupt (polled) mode:

1. Write $7F (127) to the ICR ($DD0D or 56589). This disables all interrupts from CIA 2 (U2).

2. Write $41 (65) for output mode or $01 for input mode to Control Register A ($DD0E or 56590).

3. For output mode, Timer A must be enabled; this was accomplished as part of step 2. The maximum bit rate will be just faster than 250 kHz, which is set by writing 01 to the Timer A low byte and 00 to Timer A high byte, in that order. The Control Register setting (step 2) provided for continuous square wave output from the timer, so the SR will begin clocking data on the next rising edge from Timer A.

4. Write a data byte to the Serial Data Register ($DD0C or 56588) to start sending data. Eight data bits will be shifted out, then Bit 3 in the ICR will be set. Poll this bit until it goes high, then load the next byte into the Data Register.

Input operations consist of initializing the Shift Register and polling the SP bit in the ICR. Save the input data and poll again until all data is received. As mentioned above, some I/O line could be used as a status flag or handshake.

All of our interface experiments so far have used either the C64 User Port, any RS-232 Serial Port, or the Radio Shack Color Computer printer port. All these computer inputs except the User Port are clumsy at best, leading to contrived or inefficient hardware. Expansion using the User Port is possible as discussed above, with little hardware penalty. The major tradeoffs are in operating speed and software overhead, especially for expansion beyond four 8-bit ports. One important advantage in using these ports is that it is relatively difficult to bomb your computer through these ports, compared to using the expansion ports. The expansion port on most appliance computers is unbuffered, which means a slip on your part can allow you to crater the microprocessor itself, killing your computer.

Since our next type of expansion will deal with direct expansion from the microprocessor bus, we need to discuss ways to avoid damage to the computer. Unless specific machines are mentioned in the context of hardware design, comments in future columns and the remarks to follow will apply equally to the Color Computer, VIC-20 and C64. Many will be applicable to the Apple, and possibly to the Atari computers. However, I have no documentation on Atari, and Apple expansion using the Peripheral Connectors is a detailed and complicated process.

Successful interfacing to a microprocessor bus involves a detailed understanding of bus timing, bus drive capability and characteristics of the devices connected to the bus. I'll take it easy on the details, but there will be a lot of explanation which you need to follow. Figures 4 & 5 show the two major *machine cycles* of the 65xx microprocessor - one of this family is used in C-64, VIC-20, Apple and Atari computers. Note that these two cycles are almost identical, except for the phase of the R/W* (READ/WRITE NOT) waveform. In both cases, the action takes place during the last half of the cycle. Early in either cycle the Address lines come up with the memory address being accessed, and either READ (Figure 4) or WRITE* (Figure 5) comes true about the same time. In the READ cycle (Figure 4), data comes from the memory or peripheral (such as PIA or CIA) and must be available for a *minimum* of time T4. That is, the 65xx microprocessor is guaranteed to capture data available within that time frame. You would need to make such a study when choosing memory devices or designing hardware to use with the processor. Similarly, Figure 5 shows that the 65xx processor is guaranteed to make data available for memory or peripherals *no later* than T4 seconds after tADDRESS and R/W* comes true. These times are important when designing peripherals for the processor family.

The other times shown in Figures 4 & 5 are: T1 - the maximum time required for ADDRESS and R/W* to come true; T2, T3 & T5 - the minimum time each signal will be available after the end of the current clock cycle (one full clock cycle shown). This kind of design study is called *worst case* design, because those times most likely to cause a circuit failure are chosen from the manufacturer's data sheets.

The following table lists the times corresponding to the T times in Figures 4 & 5 for a processor clock rate of 1 MHz. At 1 MHz, the machine cycle is one microsecond (1000 nSec) long. For slower clock speeds on a 1 MHz rated processor, the times will be approximately the same, which allows more time for the hardware to deliver or accept data.

TIME	Figure 4	Figure 5
T1	225 nSec	225 nSec
T2	30 nSec	30 nSec
T3	30 nSec	30 nSec
T4	650 nSec	150 nSec
T5	10 nSec	10 nSec

It will be helpful if you keep this latter portion of this column handy for reference during future columns, since this basic information will be needed for reference as you read some future columns.

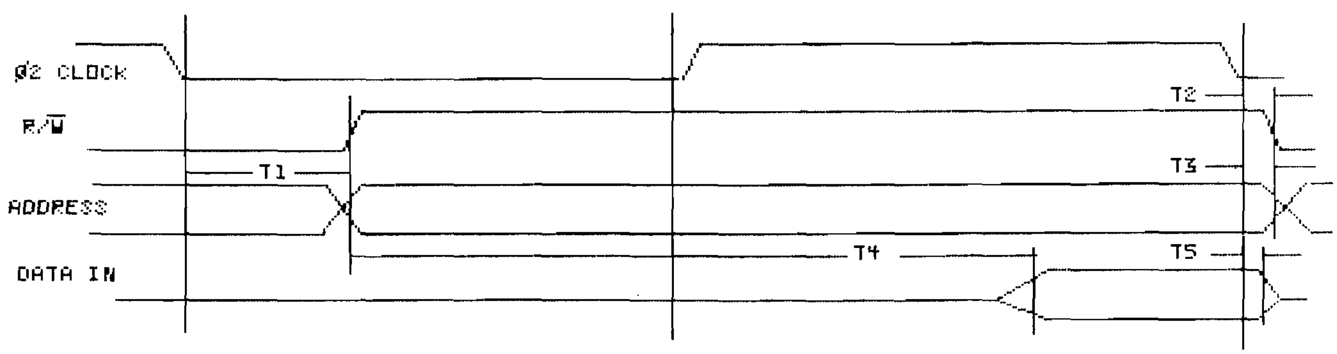


Figure 4. Timing for the 6502 READ machine cycle. See text for details of operation and clock speed.

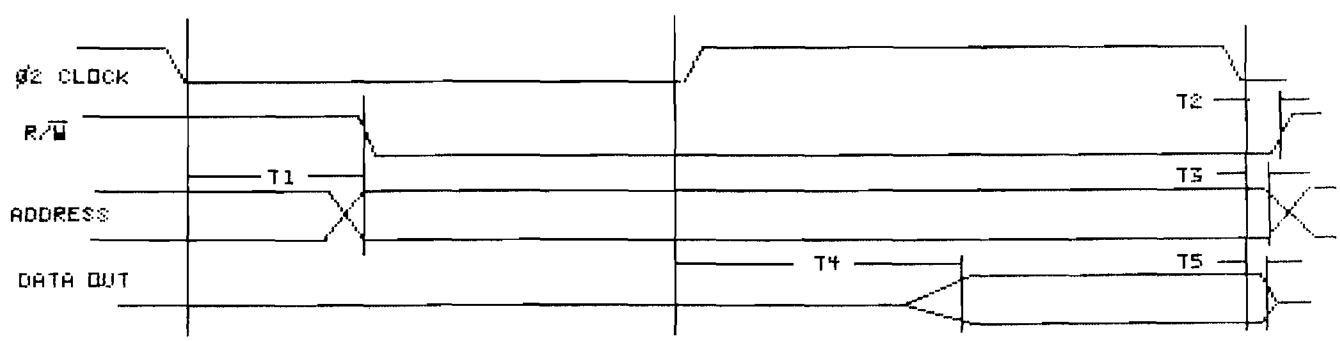
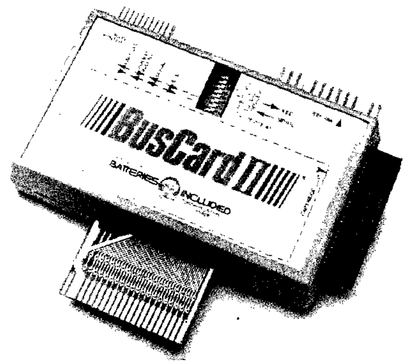


Figure 5. Timing for the 6502 WRITE machine cycle. See text for details of operation and clock speed.

Name: **BusCard II**
System: Commodore 64

Description: Allows any Commodore-compatible disk drive, including hard disk, and virtually any printer to be added to your system. Mix and match peripherals with no fear of software incompatibility. BusCard is both hardware and software invisible. The cartridge mount allows cartridges to lie flat and device allocation switches remain in function mode at all times.

BusCard gives the added power of extended BASIC as well as selectable conversion of Commodore code to standard ASCII. It comes with a full machine language monitor including assemble/disassemble commands. It just plugs in to install and comes with a one year warranty and documentation.



Price: $200
Contact: Batteries Included
186 Queen St. West
Toronto, Ontario M5v
1z1 Canada
416/596-1405

Name: **The Consultant**
System: Commodore computers

Description: Formerly Delphi's ORACLE, this program lets you design a "layout" for any kind of information you want to file, then allows you to search, sort, and analyze information. The file structure offers expandable record size up to 9 display pages (7,000 characters), with up to 99 fields per record. Any field may be a key-field, and a single field can be an entire screen of information. The number and overall size of the files is limited only by disk storage capacity. Sorting and searching is almost unlimited, including multiple-field and wild card in non-keyed fields. Full Four-function arithmetic is included. There is a password security system. Output functions include page numbering, printer control characters and optional line length. There are also built-in routines for mailing labels and forms.

Price: $125
Contact: Batteries Included
186 Queen St.West
Toronto, Ontario M5V
1Z1 Canada
416/596-1405

Name: **BLAST (Blocked Asynchronous Transmission)**
System: Over 60 micros, etc.
(not for Atari/Commodore)

Description: Asynchronous communications software which allows any computer with BLAST to talk to any other computer with BLAST, using any asynch modems, or directly linked at speeds from 300 to 19,200 baud. This package will already run on more than 60 micros, minis and mainframe systems. Unlike earlier asynchronous software, this provides truly bidirectional operation, allowing a system to receive one file, while simultaneously sending another. BLAST operates through common RS-232 serial ports and asynchronous modems, over dial-up lines or private networks, as well as from port to port. It is menu driven, supports unattended operation, permits user-defined function keys,etc.

Price: $250 (for micros)
Contact: Communications
Research Group
8939 Jefferson Hwy
Baton Rouge, LA 70809
504/923-0888

Name: **MasterFORTH**
System: Apple II/II+/IIe
Language: FORTH

Description: A complete professional programming language which includes a built in macro-assembler with local lables, a screen editor and a string handling package. The input and output streams are fully redirectable and make full use of the Apple DOS 3.3 file system. Floating Point and HIRES options are also available. This meets all provisions of the Forth-83 International Standard.

The package includes FORTH Tools, a 200 page textbook, a technical reference manual, and a complete listing of the MasterFORTH nucleus. It gives the user a view of input and output from reading the input stream to writing a mailing list program. Numerous examples are provided.

Price: $100-$160
Contact: MicroMotion
12077 Wilshire Blvd. Ste 506
Los Angeles, CA 90025
213/821-4340

Name: **The Print Shop**
System: Apple II + /Apple IIe
Memory: 48K
Hardware: Popular printers such as Epson, Imagewriter, Apple Dot Matrix, C.Itoh

Description: Write, design, and print your own greeting cards, stationery, letterhead, signs and even banners. No special knowledge of graphics is required for this menu-driven software. With keyboard or joystick you can produce a finished piece in one of eight different typestyles, in two sizes and in solid, outline and three-dimensional formats. There are nine border designs, ten abstract patterns, and dozens of pictures and symbols with which to create. A built-in graphics editor allows you to create your own pictures or modify those provided. You can also print work generated with other graphics programs.

Text-editing features such as automatic centering, left and right justification and proportional spacing give added help in design. Comes with a colorful assortment of pin-feed paper and matching envelopes, and a reference manual.

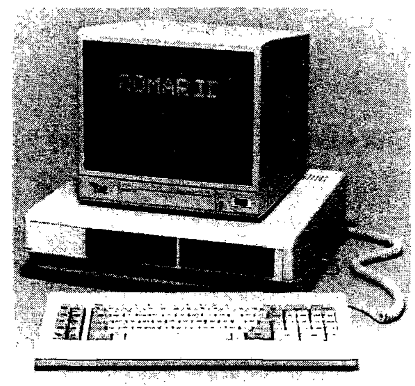
Price: $49.95
Contact: Broderbund Software
17 Paul Drive
San Rafael, CA 94903
415/479-1170

Name: **Romar II(x) Computer**
Memory: 64K (expandable to 192K)

Description: An Apple compatible computer with detached keyboard and dual capability featuring both Apple DOS and CP/M operating systems.

Based on a 6502 with 64K ROM, expandable to 192K, plus a Z080 circuit card for running CP/M programs. The separate fully encodedkeyboard contains 87 keys, including both special function and numeric keypads with CAP LOCK and status keys. Built-in command software allows most keys to be pre-programmed for special functions. The design accommodates dual floppy disk drives and an 80 watt switching power supply. It contains eight expansion slots for Apple accessories and add-ons.

The computer is designed to work with Apple programs and accessories without infringing on Apple proprietary circuitry or ROM. Besides operating with a variety of today's languages, it can adapt to future software languages.



Price: $695
Contact: Romar Computer Systems
22110 Clarendon St., Ste 103
Woodland Hills, CA 91367
818/999-1083

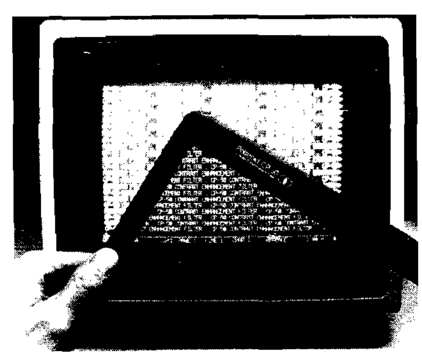
Name: **Video*Clear Interference Cable**
System: Commodore, Radio Shack CoCo, any with TV monitor

Description: An Interference Rejection Cable designed to reduce or eliminate radio frequency interference in television sets that are being used as monitors for home computers. It is easy to install, requires no modifications to either the computer or TV, and comes with all adapters necessary to interface with a wide variety of TVs. It contains a special RF filter that is designed to reduce the level of interference generated by the computer. The cable comes with 90-day warranty.

Price: $16.95
Available: Computer Associates
Box 683
West Fargo, ND 58078

Name: **PerfectView**
System: Virtually All

Description: Designed to fit virtually all terminals, this is an effective, simple and cost-efficient computer screen filter with circular polarization. It improves user comfort with glare reduction and contrast enhancement, cutting eyestrain and fatigue. It is lightweight and durable, anti-reflective coated polyester laminated to a circular polarizer. PerfectView is available in five screen sizes and mounts to the CRT housing with no tools. This is manufactured by Polaroid Corp.



Price: $49.95
Contact: PerfectData Corp.
9174 Deering Avenue
Chatsworth, CA 91311
213/998-2400

books

Title: Microcomputer Communications - A Window on the World
Authors: Barbara E. McMullen and John F. McMullen
Price: $14.95
Publisher: Wiley Press

Written in an easy, readable style, Microcomputer Communications is a guide for using your personal computer as a telecommunications tool. Methodology, equipment, and the process of information transmission is explained. It gives the essential information that is necessary for setting up telecommunications links between micros and such services as CompuServe, Dow Jones and other information sources. Telenet, Tymnet and Uninet telephone numbers for across the United States are also provided.

Level: beginner to intermediate.

Title: Engines of the Mind - A History of the Computer (hardbound)
Author: Joel Shurkin
Price: $17.50
Publisher: W.W. Norton & Company

This book presents a history of the computer from 'the mad genius of Charles Babbage and the remarkable Countess of Lovelace, through the invention of the first electronic all-purpose digital machine, to the creation of the chip and beyond.' 'Engines of the Mind' covers more than machines; it is really about people. Covering the various controversies involved in the creation of the computer, the history of the computer is painted in terms of humanity rather than a list of dates and events.

Title: Commodore 64 Graphics & Sound Programming
Author: Stan Krute
Price: $15.00
Publisher: Tab Books

Through various programs the reader is instructed in how to master the graphic and sound capabilities of the Commodore 64. Written in non-technical terms the programs use BASIC to produce effects that would require assembly language on other computers. A total of 68 programs are included with many figures, charts and diagrams interspersed throughout the text. Taking a 'learning by doing' approach, each chapter has a summary and exercises. Each chapter takes a similar format: a short introduction, programming example, detailed discussion of the example, suggestions for modifying the original, short review questions and several programming exercises. Answers and possible solutions to the problems are provided.

Level: advanced beginner to intermediate.

Title: Experiments in Four Dimensions
Author: David L. Heiserman
Price: $17.50
Publisher: Tab Books

This is an introduction to fourth dimensional geometry and its applications. There are experiments that illustrate various theories and principles of one, two and three dimensions, as well as time, matter and space. The construction of four dimensional objects is explained through the plotting of lines, plane figures and space objects. Hyperspace objects, scaling and rotations in four dimensional space are also covered. There are ample drawings and examples throughout the text. Paper and pencil are the tools that are necessary; drawing the figures on your microcomputer is optional. For those with micros, a program is provided to enable you to draw and manipulate four dimensional objects.

Level: intermediate to advanced.

Title: Picture Perfect Programming in Applesoft BASIC
Authors: Dr. Thomas Mason, Steve Payne, and Barbara Black
Price: $14.95
Publisher: Reston Publishing Company

This book takes the approach that programming in BASIC can be learned more enjoyably through computer graphics than business or math problems. Requiring only basic math skills, the reader is guided through loops, subroutines, interactive programming, high resolution graphics, and business graphics. The basic premise the book is built on is that there are only two key concepts to mastering programming - loops and decomposition. These and all concepts are demonstrated visually, using the old adages - seeing is believing and a picture is worth a thousand words.

Level: beginner.

Title: The Microcomputer Users Handbook 1984
Authors: Dennis Longley and Michael Shain
Publisher: Wiley-Interscience

Written for a wide range of business people, this handbook address the problems of choosing and upgrading microcomputer systems. Divided into two parts, the first part explains what the role of microcomputers in business is, the right way to buy a computer system, maintenance and after sales support, planning for growth, project planning and staff participation and questions to be asked at a demonstration. The second part consists of over 200 reviews of business micros and several hundred peripherals. The workings of a computer are explained, with most every related subject (languages, operating systems, telecommunications, etc.) being touched upon. There are plans for a yearly update of this handbook to keep material abreast with the market and industry.

Level: beginner to advanced.

?question mark?

Recently I received a long letter from Paulo C., a reader in Mexico, requesting help from the other readers of Micro. The following is excerpted from this letter:

"Approximately five months ago an archaeological team from the University of Mexico at Quihexl made a startling discovery while digging at the base of a pyramid in Teotihuacan. As you may know, this site has long been regarded as an area rich in artifacts and relics. This particular expedition, headed by Drs. Jose Ferra and Juan Cortese sought to discover a burial chamber in the lower levels of the pyramids, in hopes of finding similarities to the pyramids at Giza, Egypt.

After excavating an area 10' by 12' to a depth of 7', a solid stone table was struck. At first thought to be a fallen slab, it was soon realized that the stone was the top of an entrance way. Further digging revealed a wooden door, covered and sealed with gold panels. It was carefully examined by Drs. Ferra and Cortese and determined to be authentic. After being removed, the door was shipped to the Museum at Mexico City where it underwent carbon 14 tests, its age -- the same as that of the Egyptian pyramids. Following removal of the door, the team worked to clear the passageway of dirt and rocks; at last entrance was gained. The inside walls were polished smooth and the floor was made of smaller stones, cut and laid cobblestone fashion. Travelling some 8 feet down the passageway another door, similar to the first, was found tightly sealed. This door was also removed and shipped back to the Museum where it underwent many tests, including a carbon 14 confirmation. Behind it, a wider corridor ran about ten feet, then turned sharply to the right and quickly narrowed. It is perhaps best to directly quote Dr. Cortese's description of the next part of the expedition - 'There was much excitement as we walked around the corner of the corridor. When I saw that it narrowed I was particularly excited as this was the construction used in Giza preceding a staircase. So, I was not surprised to indeed find a staircase at the corridor's end.

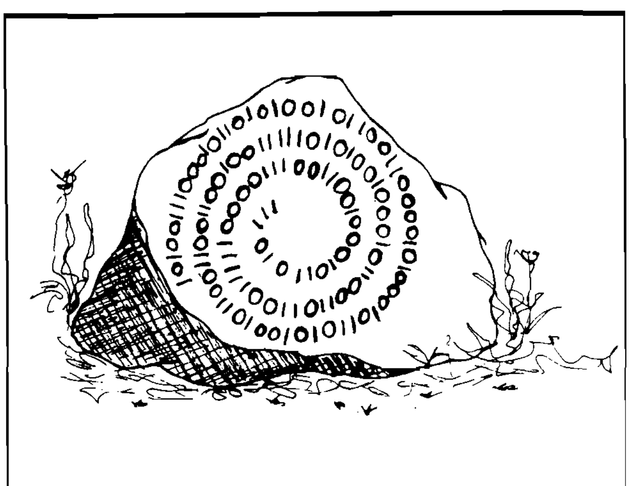
With great anticipation I slowly descended the steps, almost falling once as the generator faltered, momentarily flickering the lights. Could some angry, disturbed spirit be at work? I continued with Dr. Ferra close behind me. As we cautiously descended, we could hear a faint rustling, rushing noise, becoming louder the further we went. I shone my lantern down the stairs and, unexpectedly, it was reflected back. We reached the bottom to discover rushing water flowing over the last steps and filling the connecting passages within a few feet of the top. With the depth measured at over five feet, it was obviously too deep to negotiate without a boat, especially given the strength of the current. Alas, we had to resign ourselves to going back for the time being.'

As the team wandered back up the stairs and returned to the campsite, it was decided that canoes would be the best solution to exploring the water-filled passageway, being light and easily maneuvered. Arrangements were made to have two canoes sent immediately by the University. Two days later, equipped with canoes, lanterns, photographic equipment and excavation tools the team once again descended the stairway. The canoes and equipment were carefully placed in the water, with Drs. Ferra and Cortese in the first canoe and two of their assistants following in the second. The rest of the team remained behind. Again Dr. Cortese comments, 'We carefully got in the canoes so as not to upset them and lose our equipment. Drifting with the current, we noted an end to the passageway some forty yards further on. As the current slowed near the end, an arch top became visable. Ducking our heads low, with a few good strokes of the paddles we were through the end and out the other side. It is so hard to describe moments like this. Dr. Ferra and I both gasped. We had found the burial chamber. The ceiling was quite high, particularly when you considered we were already elevated five feet by the water. As our assistants entered, their extra lanterns illuminated the chamber more clearly and they expressed equal astonishment. We were most impressed by what appeared to be a large altar rising up out of the water against one wall, being somewhat reminiscent of the old Catholic wall altars. Next to it, on the left, was a pedestal with a large falcon type bird carved out of what appeared to be black onyx. It looked very much like the Egyptian god Horus. On the altar top, leaning against the wall, was the most interesting object of all. A large, carved, circular stone; Dr. Ferra's first impression was that of the Rosetta stone.'

The two canoes returned, cameras filled with pictures of the chamber. The exploration and retrieval of the artifacts proceeded well, the wheel being removed and sent back to the Museum. It is still under investigation and being subjected to further testing. Underwater divers photographed a carving on the front of the altar. It depicts an ancient warrior who is sitting above an illuminated rectangular box.

Unfortunately at this time there isn't any more information regarding the stone. After examining a drawing of this stone, I felt perhaps one of Micro's readers could possibly ascertain its hidden meaning. Included for you is a rendering of the stone's carvings."

Well readers, good luck and please send any theories to me, as Micro would like to be able to contribute to, if not provide, the solution. We will acknowledge whoever is the first to solve, or lead to the solution of, this mystery. Thanks; if anyone can do it, I'm sure it will be one of you.



MICRO Program Listing Conventions

Commodore

LISTING	C64 KEYBOARD
Commands	
{CLEAR}	^ CLR
{HOME}	HOME
{INSERT}	^ INST
{DOWN}	CRSR DOWN
{UP}	^ CRSR UP
{RIGHT}	CRSR RIGHT
{LEFT}	^ CRSR LEFT

Colors

{BLACK}	CTRL 1 BLK
{WHITE}	CTRL 2 WHT
{RED}	CTRL 3 RED
{CYN}	CTRL 4 CYN
{PURPLE}	CTRL 5 PUR
{GREEN}	CTRL 6 GRN
{BLUE}	CTRL 7 BLU
{YELLOW}	CTRL 8 YEL
{RVS}	CTRL 9 RVS ON
{RVSOFF}	CTRL 0 RVS OFF
{ORANGE}	= 1
{BROWN}	= 2
{GREY 1}	= 3
{GREY 1}	= 4
{GREY 2}	= 5
{LT GREEN}	= 6
{LT BLUE}	= 7
{GREY 3}	= 8

Functions

{F1}	f1
{F2}	^ f2
{F3}	f3
{F4}	^ f4
{F5}	f5
{F6}	^ f6
{F7}	f7
{F8}	^ f8

Special Characters

{PI}	π ^ Pi Char
{POUND}	£ Pound Sign
{UP ARROW}	↑ Up Arrow
{BACK ARROW}	← Back Arrow

Atari

Conventions used in ATARI Listings.

Normal Alphanumeric appear as UPPER CASE:
SAMPLE

Reversed Alphanumeric appear as lower case:
yES (y is reversed)

Special Control Characters in quotes appear as:
{command} as follows:

Listing	Command	ATARI Keys
{UP}	Cursor Up	↑ ESC/CTRL -
{DOWN}	Cursor Down	↓ ESC/CTRL =
{LEFT}	Cursor Left	← ESC/CTRL +
{RIGHT}	Cursor Right	→ ESC/CTRL *
{CLEAR}	Clear Screen	ESC/CLEAR
{BACK}	Back Space	ESC/BACK S
{TAB}	Cursor to Tab	ESC/TAB
{DELETE LINE}	Delete Line	ESC/SHIFT DELETE
{INSERT LINE}	Insert Line	ESC/SHIFT INSERT
{CLEAR TAB}	Clear Tab Stop	ESC/CTRL TAB
{SET TAB}	Set Tab Stop	ESC/SHIFT TAB
{BEEP}	Beep Speaker	ESC/CTRL 2
{DELETE}	Delete Char.	ESC/CTRL BACK S
{INSERT}	Insert Char.	ESC/CTRL INSERT
{CTRL A}	Graphic Char.	CTRL A
	where A is any Graphic Letter Key	

Non-Keyboard Commands

{DIS=}	CHR$(8)
{ENB=}	CHR$(9)
{LOWER CASE}	CHR$(14)
{UPPER CASE}	CHR$(142)
{^RETURN}	CHR$(142)
{DEL}	CHR$(20)
{SPACE}	CHR$(160)

Notes:

1. ^ represents SHIFT KEY
2. = represents Commodore key in lower left corner of keyboard
3. CTRL represents CTRL Key
4. Graphics characters represented in Listing by keystrokes required to generate the character
5. A number directly after a {SYMBOL} indicates multiples of the SYMBOL: {DOWN6} would mean DOWN 6 times

Advertiser's Index

Next Month In MICRO

Features

The Dvorak — A look at the alternative to the standard QWERTY keyboard, one which can speed up your typing unbelieveably.

Compression — This professional quality Applesoft program will save large amounts of storage space by eliminating unnecessary data and redundancies.

HiRes Printer Dump — The beginning of a three-part series on dumping graphics from the Commodore 64.

Flight Simulator — An in-depth look at the new Flight Simulator II program for armchair pilots which includes a WW II dogfight, as well as realistic instruction in flying a Piper.

Hilister — The first of a two-part series covers the highlighting of text within a program for emphasis. Part 2 will cover moving around within a program listing.

Commodore to Apple Cassette — Transfer files from Commodore to Apple and learn, in detail, how the Commodore tape is generated.

Plus More...

Departments

Reviews in Brief
Software/Hardware Catalogs
New Publications
Interface Clinic

ATARI COMPUTER OWNERS:

Pick the positively perfect, practical, printer-port peripheral package, from PERCOM DATA!

That's right... the positively perfect PERCOM DATA 5¼", floppy disk drive with a BUILT-IN PRINTER-PORT, for your Atari® 400/800 is now available!

Until now, Atari computer owners who wanted to hook a printer to their computer had only one choice... spend about $220 for an interface device. THOSE DAYS ARE OVER. PERCOM DATA has built a parallel printer-port right into its new AT88 PD model. Now you can add a quality disk drive system AND have a place to plug in a printer... WITHOUT BUYING an interface.

The AT88 S1 PD™ disk drive operates in both single density (88K bytes formatted) and double density (176K bytes formatted).

What more could you want? NO INTERFACE... a high quality PERCOM DATA disk drive... AND a built-in PRINTER-PORT... all with a price of $599.

Pick up a positively perfect PERCOM DATA disk drive, with printer-port... pronto!

For the name of an authorized PERCOM DATA Dealer near you, call our **TOLL-FREE HOTLINE 1-800-527-1222 NOW,** or write for more information.

Perfectly Priced

$599.

PERCOM DATA
CORPORATION

Expanding Your Peripheral Vision

DRIVES • NETWORKS • SOFTWARE

11220 Pagemill Road, Dallas, Texas 75243 (214) 340-5800
1-800-527-1222

Prices subject to change without notice.