SuperPET Gazette

Ave et Vale... We regret to announce that this is the next to the last issue of the SuperPET Gazette, not from choice but from necessity. The membership in ISPUG for a time exceeded 1200--but, after the end of production of SPET was announced in these pages, it slowly began to erode. New memberships no longer compensated for those we lost. We sharpened our pencil and calculated the point at which we could no longer support the overhead of office rent, a telephone at business rates, the costs of correpondence and long-distance calls, equipment maintenance and all the rest. The figures told us we'd go broke quickly with a membership of less than 500. We watched the membership figures slowly decline, held our breath, and hoped for the best. Alas...

Renewals and new memberships haven't arrived in sufficient numbers. We must now plan to cease publication after the June/July issue. We'll carry those whose subscriptions expire with this issue to the next and last one; please do not send in any more renewals.

The last issue will provide a list of all existing SuperPET software from ISPUG. We'll continue to provide disks and other ISPUG material so long as our SuperPET stands up, but from the trouble we've encountered in obtaining service, we can't be optimistic on how long that will be. One of our disk drives has been absent for maintenance for four months; it remains useless.

After the June/July issue is published, we'll refund the excess membership fees to each of you. We ask patience on this, because we'll have to write a program to screen the mail list, determine what is owed, and write the checks. Please do not ask us to swap what is owed on unexpired memberships for disks and other material. We'd go quite literally mad trying to keep such accounts by hand or on an individual basis. Five hundred doesn't sound like much until you print 500 mailing labels. Such a paper dragon's tail is 63 feet long!

Last Chance for Schematics

> Schematics for both the 2- and 3-board SuperPETs are offered for the last time, for $11 U.S. Any orders must be received by June 10. That day we'll order a specific quantity from our print shop. Order NOW if you want them! We will ship about June 20; be patient, please.

Change for the sake of change is the state of the present world. We can no longer call it progress, but an inward spiral into greater complexity and cost with questionable compensations. Manufacturers who fail to make what the market expects cannot survive. Today, the market expects 16- and 32-bit computers, icons, mice, graphics, networking, and multitasking, and that is what manufacturers make. Despite that fact that SuperPET for many of us does all the work we want done, the machine is no longer supported by maintenance or parts. When a SuperPET becomes seriously ill, where is the remedy?

Because we enjoy computing and writing about computers, we're going to take a shot at a journal for the Amiga, called The Amigan Apprentice & Journeyman. We hope it brings us as many friends as has the Gazette. You have been a splendid group to work with; witty, patient, clever, intelligent, enjoyable--and forbearing. If any of you care to voyage with us upon Amigan seas, we'd love to have you with us. Send a buck for postage only; we'll return the first, 54-page issue of the Amigan A&J.

ON THE WOLVES OF THE PRESS
or,
GOOD NEWS IS NO NEWS

Man and boy, we have watched the wolves of the American press rend to shreds anyone who showed the slightest vulnerability--a wound, an indiscretion, the merest trace of blood. The press delights in destruction. In that press, of late, we have read shrill cries that Commodore may soon be bankrupt or seek protection from its creditors by going to Court for protection under Chapter 11. Herewith, some facts:

In the fourth quarter of 1985, Commodore sold over one million computers. Its revenues from those sales were $339.2 million--an all-time record in revenues for Commodore for any quarter. Without writedowns for inventory and the closing of plants, operating profits were $1.03 million.

Now, we hear from the rumor mill that for many a year, all those busted Plus 4s and buggy C64s returned from the toy shops were stuffed into warehouses and marked as "inventory". We won't point a finger at Kindly Uncle Jack Tramiel, or at any of the folks who sold their stock when they departed Commodore a year or so back. (Did the practice keep the price of the stock up for a while?)

Anyway, with the end of production on the 8032, the Vic, SuperPET, Plus 4, and a bunch of printers and peripherals, Commodore wound up with excess plant and useless inventory--which it is selling or writing off the books. (Have you seen the television ads for this great computer, the Plus 4, run by whoever bought them?)

To restore order, Commodore's new managers in the last quarter of 1985 wrote off a plant in England and a semiconductor plant in California for $22 million, plus $29 million more for obsolete inventory.

Writedowns obscure true operating results--and set the stage for excellent earnings. The figure to watch is cash flow--how many megabucks are coming in. Debt you service with cashflow; writedowns don't cost you a cent, nor do they threaten your creditors (the stockholders feel the pinch). Commodore is doing fine on cash flow; you can bet your socks that the the banks to which Commodore owes money won't push the company into bankruptcy so long as the golden tide flows in. So much for the yapping bad-news hounds of the press (yup, that's a pun).

A week after writing this, we found a tiny squib in InfoWorld noting that Commodore has reached an agreement with its banks for a $135 million line of credit which runs through March 15, 1987 (yes, 1987). We did not subsequently see any 24-point headlines in TIME or in any of the other sheets which had splattered the bad news over their pages. Chalk up one more failed sortie by the wolves of the media to hamstring and devour a firm that showed a trickle of blood.

ONCE OVER LIGHTLY
Miscellany

An ISPUGger equipped with an 8023 printer went to enormous efforts to indent hard copy output. As he says, "If you do not know how to tab your 8023 from the editor to make the beast starting printing on a particular column, here is a way to slide text over that I plagiarized from page 52 of the System Overview Manual" (see the monster at left, where the number of spaces preceding the '%&' determines how many spaces are inserted at the start of a line). Gee, we almost awarded him a prize for persistence and valor, but instead we broke down and wept, because it is so very easy to do the same thing with the simple critter at the left, which merely

```
*/%./c/%^%.%*%$/      %&/
```

*c/%^/   /   says to change the start of every line (%^) to begin with number of spaces between the two slashbars. Any text in an editor is then indented by that number of spaces. If you are careful not to edit further (any editing destroys invisible text past the screen right margin), text sent to printer will be complete and indented neatly. Ah, well; the problem would not exist if he could tab his 8023 printer from an editor. We don't have an 8023. Anybody know the commands? If we get 'em, we can send 'em fron BEDIT by hook or crook.

THE THREE-BOARD BUG IN OS9 We reported a few issues back that the upper 64K of memory no longer died in OS9 when that operating system was installed on some old 3-board SPETs. Seems we were wrong. Bob Davis reported that his machine wouldn't run OS9, though everything else worked fine; then we got our OS9 board hooked up, and it wouldn't work either. All installations of OS9 on two-board machines seem to be working fine; we've had nary a trouble report.

The December, 1985 issue of TPUG magazine carries a complete summary of the problem and of the proposed solution. For those who do not receive the magazine, we summarize. The problem: the 3-board machines simply will not run OS9, although the Waterloo languages and facilities run okay. The cause of the problem appears to be a mistake by Commodore which causes the 64K of bank-switched RAM not to be refreshed when you work permanently in those upper banks, as OS9 does. All dynamic RAM loses its memory contents if not refreshed now and then.

TPUG reports that a hardware engineer hired to find a fix has indeed found one. Two chips are needed, but are hard to come by. When they are in hand, TPUG will construct and test a prototype. When we hear the results, we'll report.

AGING BUT NOT SENILE: We do appreciate the thoughtful folk who carefully remove the address label from the backside of the application form and re-glue it on the front when they renew. But, dammit, although we indeed are getting older, we retain strength sufficient to turn an application over to read the label just where the printer placed it...

THE NATIONAL NO-REPAIR CENTER: Our old Tandom drive began to shriek and moan beyond its usual volume last fall, and then regurgitated error messages when it tried to read any disk made on another drive. We shipped it off to the National Repair Center, at 3354 Winbrook Drive, in Memphis Tennessee, 38116, as Commodore recommended, on November 15, 1985. We write this on March 15, 1986, four months later--and National still has not repaired the poor old thing. Every time we call, they say they need parts and expect them soon... We finally said to hell with it and told 'em to send the drive back. We suspect they will charge us for rent... If you expect service, go somewhere else.

For those in the Central U.S.: Ervin Dunham of Natchez, Mississippi, reports that Mid-Kansas Computers, at PO Box 506, Newton, Kansas 67114 did a nice job of repairing his SuperPET and for a most reasonable charge. Call 316 283 0208. But then again, you may not want to:

From Colin J. Campbell of Nova Scotia: "I phoned the firm that was selling $200 SuperPETS [Ed. Mid-Kansas, as above] the morning I received the last Gazette. I confirmed that (a) SuperPETs were available at that price; (b) they indeed had some in stock; (c) manuals were included; and (d) that they would ship one to me

including shipping charges. Six weeks passed. My VISA account was not debited. I phoned a second time and was informed that they could sell me a used model for $400+ but in reality had none when I originally called...#$%!!..."

GOTCHA, ARNOLD! "Gadzooks, first you give me h__l for not ordering the schematics soon enough, and now I find I must get another copy of BEDIT because I bought the first version too soon!" cries Arnold Marks of Miami. Sorry Arnold, but we learned how to hook folks on software a few years back. First, you give 'em a taste; when they are addicted after a year or so, you offer a purer stuff. We reckon we could go on and on, through BEDIT 4, if Joe Bostic weren't all tied up writing a decent editor for Amiga. And we're getting filthy rich at ten bucks per disk...

ML SPEEDUP ON DELTON'S HOME ACCOUNTING: A few months back, Delton B. Richardson of 4299 Old Bridge Lane, Norcross, Georgia 30092 told us that he had most considerably speeded up his Home Accounting software--which some of you may have bought from him for $15 U.S. (you can order the improved version for the same price directly from Delton. Don't write us!. All versions are for 8050 format only). He sent us a sample disk; supercharged it is! The slow stuff he redid in machine language. He also arranged to load the ML routines automatically from microBasic, so the user does not have to to fuss with it. When you use the program, you'd never know the ML routines were there, except for the speed. But--therein lies a tale...

When we got Delton's disk, we noticed a lot of PEEKery and POKEry going on as Delton loaded the ML stuff; we learned by inquiry that the black magic simply reset MemEnd_ and recreated the pointers and data mBasic needs just below that new MemEnd_. So we asked Delton if he had re-invented the wheel, for we had published in Issue 13, Volume I, page 219, a simple method to do just that. Delton, sobbing, reported that "I have in fact re-invented the wheel! This is a rather common problem in software...". Indeed it is, and we blame ourselves for entitling the piece "To Kill a Mockingbird"; which, if highly literary and allusive (we are addicted to both), is hardly descriptive. Sorry, Delton. You can write that week of work off to our fondness for cryptic and mysterious titles.

ANYONE FOR TENNIS? It takes about two days to create a cross-referenced index to a Volume of the Gazette. We haven't had time to do it for Volume II, and ask for a volunteer to create one. Every major subject must be entered under a couple of logical index headings. Then you sort the list, edit and format it, which is easy. Because we're working with two different computers on a 36-hour day schedule, we don't have the time. If you want an index, make one! Write if you can take the job on! Please give a hand!

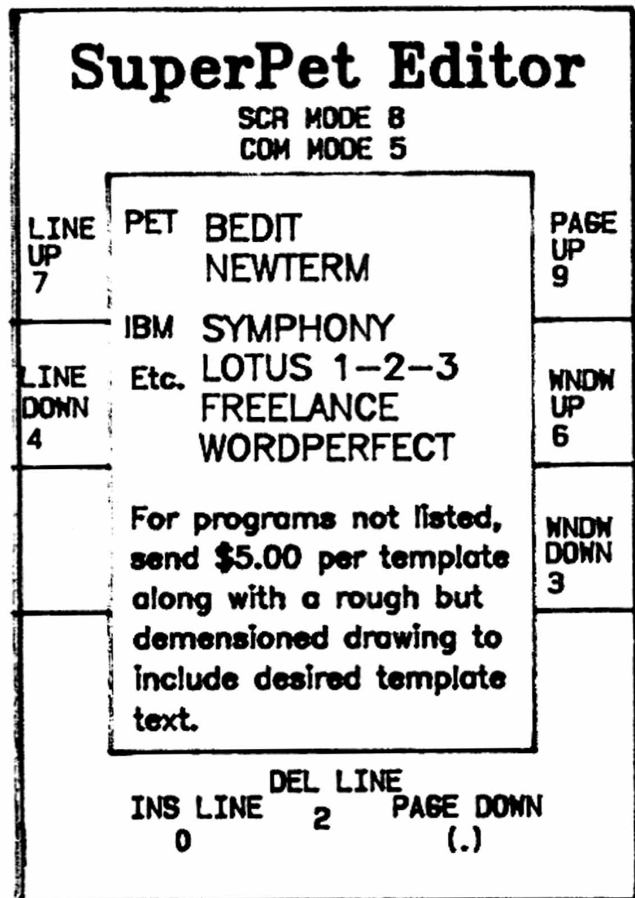
MORE ON VISICALC: A few issues back, poor Marilyn Post asked where in the world she could get a copy of Visicalc for SPET. Two kind souls replied to her, making it available, but the remaining problem is instructions. John Frost of Seattle writes that "The Visicalc Program Made Easy," by David M. Castlewitz, is available from Osborne/McGraw-Hill, and certainly will serve.

ANOTHER SPET FOR SALE: Ken Druze of 1830 Fernwood Road, South Belmar, N.J., 07719, offers a SPET, 4022 printer, SFD 1001 and 2031 drives, Visicalc, POWER, all manuals, plus schematics and a 64K memory expansion used only with Visicalc. For details: (201) 747 6745 weekdays, 10-6, or (201) 681 1353 at other hours. The rig goes as a complete package. Original boxes available for shipment.

MOSES DIDN'T... Charles Kiessling of Berkshire NY writes that "Your evaluations of Amiga are the only ones that are worth anything. Most reviews read as if Moses carried it down the mountain with the commandments." Yeah, we noticed that, too, and wondered if the other reviewers got a different model--or just didn't do their homework. Well, it is a good machine, but it doth have flaws.

BUG IN BEDIT 2: As much as Bedit 2 was massaged before release, a l'il bug is in it. Any attempt at a global change with the tilde used in the search string won't work properly (the tilde here represents any non-alphabetic character). Bedit simply ignores it. If a change command is given a line at a time, the tilde works okay. Joe Bostic is working on a fix.

TEMPLATES FOR THE KEYPAD: John Seither of 4242 Briars Rroad, Olney, MD 20832 has a new Hewlett-Packard 8-pen high-res plotter, with which he creates templates for keypads or function keys (IBM, Compaq, SPET or Televideo). Templates are in clear, flexible plastic; they fit around and identify the function of each key. While we can remember the mEDIT keys in our sleep on a black night in a hurricane, many folks can't; we find the template for NEWTERM keys very useful. We show one of the overlays at left (for the Editor). Similar templates are available for SPET's NEWTERM; for Symphony, Lotus 1-2-3, Freelance and WordPerfect on the IBM PC, Compaq, and Televideo.



The clear plastic templates are durable; better than the paper ones often furnished with programs. John wants $2.50 each for the templates listed above--minimum order $5.

If you want a custom template for any of the machines (for programs which aren't listed above), John will make them for $5 each--but only if you send him a rough but dimensioned drawing of what you want, plus the text you want on the template.

We don't know how well the image of the template will reproduce, because John used blue ink for the outline, and blue doesn't reproduce too well--sometimes. If the borders look sorta amateurish--we redid them in black.

AN EASY WAY TO PASS PARAMETERS BETWEEN LANGUAGE AND ML ROUTINES
("How Come So Previously Dumb?" Dept.)

For the past year, we have tried and have abandoned some six different ways to pass parameters between high-level languages and assembly language routines. We demanded that the method pass named variables from a HLL (high-level language) and that parms coming back from the ML routines also become named var-

iables in the language. All methods we tried were slow, complex, and specific to a particular task. We kept looking for a method which would pass any number of parms to ML routines and would return one or two from the ML routines as named variables in the language. Last, we wanted to be able to pass error messages from ML back to HLL. And we wanted to do all this with a standard routine which would be universal, requiring no change from program to program.

When we finally figured out how to do it, we blushed--the method is so simple it ought to have been invented about the time Cheops built his pyramid.

OUTLINE OF METHOD: The problem in passing parms always is their address. Now that we know how, it seems obvious that the screen and the keyboard buffer are the places to pass them, for addresses in both are easily defined. The screen has one limitation, however: you can't pass unprintable parms unless you poke them--a slow process. We therefore gave the keyboard buffer a try, and found we could pass parms back and forth with ease. Here's the approach:

1. Pass language parms by printing them into the keyboard buffer. We set the high and low pointers to the buffer to $130 (decimal 304), the start of the the buffer. We then print into the buffer the count of parms sent and the parms themselves (up to 39 characters worth). Then we SYS our ML routine.

2. The ML routine starts with a parm-getter which looks in the keyboard buffer and stuffs the parms into buffers. Parms may be data or system addresses.

3. Once the parms are stored, the ML routine calls an "execute" module which executes whatever code or system routine is wanted. As its last step, the "execute" module stuffs the results back into one or more "result" buffers.

4. A "putparm" ML routine then stuffs the result buffers into the keyboard buffer, each parm being ended with a CR. We now RTS back to language from ML.

5. The lines in language which follow the SYS call ask for input. In every language we've used, every input statement always dumps the keyboard buffer to the next CR. Well, all returned parms are stuffed in the buffer with a terminal CR. They dump at "input" just as if we'd typed them in. We automatically get named variables with values in HLL.

```
sys hex('7008')
input "",return$
input "",error$
```

There are three ML routines: a parm getter which never changes; an "execute" module which must be written by the user, and a parm putter which never changes. In mBASIC, we need only one short language module, ten lines long. It passes any number of parms and likewise need not change. Of the four modules, three are general-purpose and should work with any program. The method should work in any SuperPET high-level language.

In the programs below, we chose to pass all parms as strings, in both directions. When we pass parms from language to ML, integer or real variables appear in the keyboard with prefixed and suffixed spaces; string variables do not. When we pass parms from ML to language, we know strings will print and input okay in HLL. Consistency in data type avoids programming problems.

Last, a note on the pointers to the keyboard buffer; the first (at $012C) points to the first character entered; the second (at $012E) points one byte

past the last character entered. If both pointers point to the same address, the buffer is "empty." If the pointers are apart, the low pointer moves up to the high pointer, printing to screen all characters between the pointers at any input statement, PAUSE, or STOP. We adjust the pointers so that the buffer will hold only parms, and nothing but parms, no matter what keys the careless or idiotic may press on the keyboard while the program runs (the keyboard is disabled within the passparm procedure itself). For more details on the keyboard buffer, see Issue 1, Vol. II, p. 12ff and Vol. I, p. 122 and p. 219.

The mBASIC demo below generates parms to pass and gets back from the ML routines the answer and any error message. Note that all parms received from the ML module are "input" into the HLL as named language variables. Procedure "putparm" will pass any number of parms, and need not be revised.

```
100 ! marry:bd. mBASIC demo of parm passing. Adds two integer parms, returns
110 ! result plus error message if total lies between 32768 and 65535.
120
130 nul$=chr$(0)  : option base 1 : dim real$(5)
140 real$(1)="1234"                       ! Vary value of input strings to
150 real$(2)="5678"                       ! exceed 32767; error msg. returns.
160 open #20, "keyboard", output
170 call putparms(mat real$,2)            ! Put parms in keyboard buffer.
180 sys hex('7008')                       ! Call ML routine.
190 input "", return$                     ! Sense result in keyboard buffer.
200 input "", error$
210 print "Value of return$ is: "; return$ ! Print language variables.
220 print "Error message is: ";error$
230 stop                                  ! A general-purpose parm-passer.
240
250 proc putparms(mat parm$,parm_count%)
260   poke hex('0129'),0                  ! Disable keyboard for dumbjohns.
270   poke hex('012d'),hex('30')          ! Set low byte of buffer pointers
280   poke hex('012f'),hex('30')          ! to start of keyboard buffer.
290   print #20, value$(parm_count%);     ! Pass parm count to ML Routine.
300   for i%=1 to parm_count%
310     print #20, parm$(i%);nul$;        ! Pass strings, without leading or
320   next i%                             ! trailing spaces, ended with null.
330   poke hex('0129'),hex('ff')          ! Enable keyboard.
340 endproc
```

The ML program below is designed to load at main menu before mBASIC is loaded. The "getparms" routine will handle any number of parms so long as a buffer is allocated for each input parameter, within the buffer limit of 39 characters.

```
;passer.asm. A demonstration of parm passing between ML and language.

XREF stoi_,itos_              ; Routines convert string to integer and
                              ; integer to string.
kyptr1_ EQU $012C
kyptr2_ EQU $012E

          LDD #getparms-2     ; This menu loader sets MemEnd_, which is
          STD $22             ; at $22, and
          CLR $32             ; clears service_ for return to main menu.
          RTS
```

```
;"Getparms" gets the counted number of parms from the keyboard buffer and stores
;them. It then calls the "execute" routine to use the data, and finally calls
;"putparms" to pass the results back into the keyboard buffer.

getparms      CLR result1      ; Zero out result buffers so routine can be
              CLR result2      ; used as often as wanted.
              LDY #$130        ; Load pointer to first character in key. buffer.
              LDX #buff1       ; Point to buffer for parm 1.
              LDA ,y+          ; Load the parm count.
              SUBA #$30        ; Convert it to a counting number.
              PSHS a           ; Stack is used as decrement register.
              CLRA             ; "A" register is an offset index.
              LOOP
                LDB ,y+        ; Get parm character.
                IF EQ          ; At end of string yet?
                  STB ,x       ; YES, store end-string null.
                  DEC ,s       ; Reduce stack parm count by one.
                  BEQ thru     ; Quit; no more parms.
                  TFR a,b      ; Compute the next buffer address by
                  LSLB         ; multiplying parm count by 2
                  LDX  #table  ; and offsetting to
                  LDX  b,x     ; load proper buffer address.
                  INCA
                ELSE
                  STB ,x+      ; Store character in buffer.
                ENDIF
              ENDLOOP
thru          LEAS 1,s         ; All parms stored.
              JSR  execute     ; Now, call routine which uses the parms.
              JSR  putparms    ; Pass the results back to keyboard buffer.
              RTS

;This routine is for demonstration only. It adds the two parms passed, stores
;the results in the result buffers, generates an error message if the result
;is wrong, and puts the error message in a result buffer. Substitute for this
;your own routine, passing all parms to the RESULT buffers as strings.

execute       LDD  #buff2       ; P1 address of second number parm.
              JSR  stoi_        ; Convert it to a counting number in D.
              STD  buff2        ; And store the result.
              LDD  #buff1       ; P1 the first parm string.
              JSR  stoi_        ; Convert to a counting number in D
              ADDD buff2        ; Add the second parm. Result is in D.
              PSHS d            ; P2 the two-byte integer sum for later use.
              GUESS
                QUIF cs         ; If carry or negative flags are set,
                QUIF mi         ; we have a wrong answer, so
              ADMIT
                LDX  #result2   ; send an error message back to language
                LDY  #errmsg
                LOOP            ; in the second result buffer.
                  LDB ,y+
                  STB ,x+
                UNTIL EQ
```

```
            ENDGUESS
            LDD  #result1      ; P1 the address of result buffer, convert
            JSR  itos_         ; sum to a string value in result buffer.
            LEAS 2,s           ; ITOS_ ends the string with a null.
            RTS

;This routine returns the results to the keyboard buffer. It sends a plain CR
;for a null parm (which enters language as a null string), or it sends back
;a real parm if there is one. We have limited this routine to 2 parameters.

putparms    LDX #$130          ; Point to start of keyboard buffer.
            PSHS x
            LDY #result1       ; Get address of first parm.
            LDA #2             ; We assume no more than two result parms.
            LOOP
              LDB ,y+          ; Get one character of result.
              IF EQ            ; If null, we are at end of parm.
                LDB #$0d       ; End parm with a CR
                STB ,x+        ; in keyboard buffer.
                DECA           ; Decrement the parm count
                BEQ fini       ; until we've passed 'em all.
                LDY #result2   ; Now pass parm 2. If null, a CR only is sent.
              ELSE
                STB ,x+        ; Stuff parm character in keyboard buffer.
              ENDIF
            ENDLOOP
fini        PULS y             ; Get address of start of keuboard buffer.
            STY  kyptr1_       ; Set low pointer to start of keyboard buffer.
            STX  kyptr2_       ; Set high pointer one past last character,
            RTS                ; so buffer will dump on an "input."

errmsg  FCC "Error. Sum too big"      ;The passer.cmd file follows:
        FCB 0                         ;  "passer.cmd"
                                      ;  org $7000
table   FDB buff2,buff3,buff4,buff5   ;  include "disk/1.watlib.exp"
                                      ;  "passer.b09"
buff1   RMB 16
buff2   RMB 16    ; There is no limit on the number of input
buff3   RMB 16    ; parms except memory used for buffers and the
buff4   RMB 16    ; 39-character limit of the keyboard buffer.
buff5   RMB 16    ; Resize these buffers as required.

result1 RMB 16    ; Result buffers may be resized as needed to a limit
                  ; of 39 characters.
result2 RMB 23
```

Although we could have used one or two consolidated buffers and saved some memory, the code gets more complex and certainly is harder to follow. We opted to keep it simple. And yes, you could stuff the results of "execute" directly into the keyboard buffer without intermediate buffers; the penalty is rewriting the buffer-stuffer code for every package. We'd rather not, thanks all the same.

BETTER WAYS OUT OF THE BLUEBERRY BUSHES
And Back Home to Mother

Last issue, we showed several ways to exit from a procedure or function, back to the calling program, before the procedure itself is completed. As usual, no sooner was it published than we received a program from Dr. John Cordes (of 50 decimal digits of accuracy in floating point fame) which shows several previously undocumented, most ingenious and useful ways to do the same thing--all of them simple and sweet.

```
100 ! endproc:bd, by John Cordes
110
120 x = 3.1
130 print tab (7); "x"; tab (20);"z"
140
150 for i = 1 to 10
160   x = x - 0.2
170   call end_proc1
180   call end_proc2
190   call end_proc3
200   print tab (5); x; tab (18);z
210 next i
220
230 proc end_proc1
240   z = x
250   if z < 2 then print "A"; : endproc
260   print "B";
270   z = x**2
280 endproc
290
300 proc end_proc2
310   z = x
320   if z < 2
330     print "a"; : endproc
340     print "Ooops!";
350   endif
360   print "C";
370   z = x**2
380 endproc
390
400 proc end_proc3
410   z = x
420   if z < 2 then endproc
430   print "D";
440   z = x**2
450 endproc
```

The first way to leave a procedure at any time is shown on line 250. There, you may exit upon any condition you may state. We should have known this was possible, for we carried an article on similar exits in a FOR...NEXT loop a few issues ago.

Line 330 shows the same method of exit in an IF...ENDIF. Both Dr. Cordes and ye ed ran this program for 1000 iterations to be sure that these "premature" exits were accepted and did not muck up the stack. No problems.

The third method (line 420) you might think is a GOTO a label, but really isn't. It is simply another way to mark the end of a procedure.

	x	z
BCD	2.9	8.41
BCD	2.7	7.29
BCD	2.5	6.25
BCD	2.3	5.29
BCD	2.1	4.41
Aa	1.9	1.9
Aa	1.7	1.7
Aa	1.5	1.5
Aa	1.3	1.3
Aa	1.1	1.1

Printed above is program output for ten iterations, which certainly shows that each EXIT works as intended. We suspect the methods above will work without change in functions.

T H E   A P L   E X P R E S S   by REG BECK

Box 16, Glen Drive, Fox Mountain, RR#2, Williams Lake, B.C., Canada V2G 2P2

We thought we had licked the problem of direct access files in APL without any need for handling the access mode and status bytes in the file control block (FCB) [see the last issue of the Gazette for details]. Using the syntax in Stan Brockman's Assembly Language program, simple APL functions to read and write sectors were written using U1, U2 and B-P. These functions worked--but after a couple of sessions at the computer proved unreliable. The READ function some-

times failed and returned nothing after supposedly reading a sector. It proved necessary, after all, to clear the EOF bit in the status byte of the FCB and to clear the 3rd and 4th bits of the access mode byte. In order to do this, it was necessary to PEEK these bytes, mask them, and then poke them back into the FCB. The following functions were written for this purpose. Direct definition was used for the result-returning functions.

```
      DISPLAY 'PEEK'
PEEK:+/(-⎕IO)+⎕AV⍳⎕PEEK ω                         ⍝RETURNS AN INTEGER.
      ∇POKE[⎕]∇
[  0]    A POKE B                                 ⍝A IS THE CODE, B IS THE
[  1]   ⎕AV[⎕IO+A] ⎕POKE B                        ⍝..ADDRESS IN DECIMAL.
      ∇FCB[⎕]∇
[  0]    R ← FCB ;FPTR;X                          ⍝RETURNS THE ADDRESS OF THE
[  1]   FPTR←3+16×7                               ⍝SECOND PART OF THE FCB.
[  2]   X←1+(256×PEEK(FPTR))+PEEK(FPTR+1)
[  3]   R←(256×PEEK(X+4))+PEEK(X+5)
      DISPLAY 'AND'                               ⍝'ANDS' TWO DECIMAL NUMBERS
AND:2⊥((8⍴2)⊤α)∧(8⍴2)⊤ω                           ⍝..BETWEEN 0 AND 255.
      ∇RESET[⎕]∇
[  0]    RESET FCB2;VAR1;VAR2                     ⍝RESETS APPROPRIATE BITS IN
[  1]   VAR2←PEEK(FCB2+12)                        ⍝..THE FCB.
[  2]   (VAR2 AND 254) POKE FCB2+12
[  3]   VAR1←PEEK(FCB2+1)
[  4]   (VAR1 AND 231) POKE FCB2+1
```

Note the use of plus reduction in PEEK. This was necessary because, contrary to the information in the Waterloo microAPL manual about PEEKing a single byte, the result is not an integer but is instead a single element vector. This should be expected since the syntax for QUAD PEEK permits a vector argument of memory locations and returns a vector result. The result of peeking memory locations is converted to decimal form by looking it up in QUAD AV. The function AND converts these and the masks to binary numbers and "ands" them, then changes the result back to decimal form. RESET then uses these to adjust the FCB, not properly handled by Waterloo's DOS/ROMs. With these additions, READ and WRITE have proven reliable during several further tests.

```
      ∇READ[⎕]∇
[  0]    Z ← READ A ;FCB2
[  1]   'IEEE8+15.⍳0' ⎕TIE 15                     ⍝INITIALIZE DRIVE 0.
[  2]   ⎕UNTIE 15
[  3]   'IEEE8+2.<' ⎕TIE 2                        ⍝REQUEST DISK BUFFER.
[  4]   FCB2←FCB                                  ⍝FIND ADDRESS OF FCB.
[  5]   ('IEEE8+15.↓1 2 0 ',⍕A) ⎕CREATE 15 ⍝SEND U1 COMMAND.
[  6]   RESET FCB2                                ⍝RESET FCB.
[  7]   Z←⎕GET 2 256                              ⍝READ 256 BYTES.
[  8]   ⎕UNTIE ⎕NUMS
[  9]   'IEEE8+15.⍳0' ⎕TIE 15                     ⍝TURN OFF DRIVE LIGHT.
[ 10]   ⎕UNTIE 15
      ∇WRITE[⎕]∇
[  0]    A WRITE TEXT
[  1]   'IEEE8+15.⍳0' ⎕TIE 15
[  2]   ⎕UNTIE 15
[  3]   'IEEE8+2.<' ⎕CREATE 2                      ⍝ALLOCATE BUFFER.
```

```
[ 4]   'IEEE8+15.⊥+* 2 0' ⎕CREATE 15        ⍝POINT AT 0 BUFFER POSITION.
[ 5]   (255↑TEXT) ⎕PUT 2                    ⍝SEND TEXT TO BUFFER.
[ 6]   ('IEEE8+15.↓2 2 0 ',⍕A) ⎕CREATE 1    ⍝SEND U2 COMMAND.
[ 7]   ⎕UNTIE ⎕NUMS
[ 8]   'IEEE8+15.ι0' ⎕TIE 15                ⍝TURN OFF DRIVE LIGHT.
[ 9]   ⎕UNTIE 15
```

The correct syntax for these functions is, for example; READ 18 1 to read track 18 sector 1 and 6 3 WRITE 'NOW IS THE TIME, ETC.', which writes the text to track 6, sector 3. You might have noticed that WRITE writes 255 bytes and READ reads 256. WRITE sends 255 bytes followed by a carriage return. All 256 bytes may be successfully READ but if an attempt is made to WRITE 256 bytes, one byte is lost. The system writes in the carriage return as the 256th byte. Attempts to suppress the carriage return and let 256 bytes be written have so far failed.

Some machine language routines executed using QUAD SYS might come in handy for direct access filing or other purposes. We first examine the form of QUAD SYS.

```
A ⎕SYS C    ⍝C IS EITHER A MEMORY LOCATION EXPRESSED AS A DECIMAL
INTEGER BETWEEN 0 AND 65535 OR A CHARACTER VECTOR OF BYTES TO EXECUTE DIRECTLY.
A IS AN OPTIONAL PARAMETER LIST TO PASS TO THE MACHINE LANGUAGE ROUTINE. THE
TWO-BYTE VALUE OF THE FIRST PARAMETER IS PASSED TO THE 6809 D REGISTER. THE
REMAINDER ARE PUSHED ON THE STACK (2 STACK BYTES FOR EACH PARAMETER). A RESULT
IS RETURNED FOR EACH ⎕SYS AND REPRESENTS THE VALUE IN THE D REGISTER AT THE
TIME OF RETURNING FROM THE MACHINE LANGUAGE ROUTINE. EXAMPLES FOLLOW:
      ⎕SYS 55721     ⍝55721 IS THE LOCATION OF A ROUTINE WHICH CLEARS THE
                     ⍝..SCREEN. A RESULT OF 160 WAS RETURNED.
    A←⎕SYS 55721     ⍝THE RESULT IS RETURNED IN A AND NOT PRINTED TO SCREEN.
```

Before we can directly execute a machine language program we must first obtain a listing of it in decimal, which we convert to characters using QUAD AV. As a simple example to illustrate the method we will use the one found on page 17 of the Waterloo 6809 Assembler manual (we replace the SWI command with the RTS command required by APL, and write in the address of PUTCHAR_):

Hexadecimal ML code	Decimal Code	Description
C6 61	192 97	Load B register with "a"
BD B0 BD	189 176 189	Jump to "PUTCHAR_" subroutine
39	57	RTS (return from subroutine)

The routine "PUTCHAR_" (for those not up on Assembly language) is a machine language routine residing in ROM in SPET's memory which will print one character to the screen. The ROM address in Hexadecimal is B0BD. To use this routine, the character to be printed is loaded into the D register (really into B, which is the lower half of D). The routine then picks the character out of the register and prints it to the screen.

```
R←⎕SYS ⎕AV[⎕IO+198 97 189 176 189 57]
A
THE WHOLE PROGRAM IS WRITTEN IN 'MACHINE LANGUAGE' AND EXECUTED DIRECTLY.
ANOTHER METHOD WOULD BE TO PASS THE LETTER 'A' AS A PARAMETER DIRECLTY INTO
THE D REGISTER (WHICH ACTUALLY PUTS IT INTO THE B REGISTER) AND THEN DIRECTLY
EXECUTE THE REST OF THE PROGRAM.
R←97 ⎕SYS ⎕AV[⎕IO+189 176 189 57]
```

```
A
57 IS THE RTS CODE WHICH MUST END EACH MACHINE LANGUAGE PROGRAM.
OF COURSE, SINCE WE KNOW THE ADDRESS OF 'PUTCHAR_',45245, WE CAN USE THE OTHER
FORM OF ⎕SYS.
R←97 ⎕SYS 45245
A
```

Because QUAD SYS is result-returning we can define a function SYS in direct definition. We will also need a function to convert Hexadecimal to decimal, as the program listings and memory locations we can find are in hex.

```
      DISPLAY 'SYS'
SYS:⎕SYS ⎕AV[⎕IO+ω]          ⍝NO OPTIONAL PARAMETER LIST.
      DISPLAY 'HTOD'
HTOD:16⊥-⎕IO-'0123456789ABCDEF'⍳⍉ω
      HTOD 'B0BD'            ⍝HEX ADDRESS CONVERTED TO DECIMAL ADDRESS.
45245
      HTOD 'B0'              ⍝CONVERSION OF ADDRESS TO HIGH-BYTE, LOW-
176                          ⍝..BYTE FORM.
      HTOD 'BD'
189
      HTOD 'C6'              ⍝CONVERSION OF JSR OP CODE.
198
      R←SYS 198 97 189 176 189 57
A
```

Now, on to the greater task. We can directly execute PIC (position independent code) using QUAD SYS, or the function SYS, previously defined. [Ed. Three great cheers! Someone has now recognized the fantastic advantages of PIC in APL, where you cannot lower the top of memory for an ML package without making all APL workspaces incompatible! Great work, Reg! You've burned off the slave chains!] Now, write a program in Assembly Language using the DEVELOPMENT package (or type one in from an example given in a previous issue of the Gazette, as we did here). Assemble it and load in the .B09 file using the APL function CONV_B09, which follows. The interested reader should read the background material on this which appeared in Vol. II, No. 2, pp 38-47. We assembled the PIC program example in the reference, creating a file, MLDUMP.B09. The program CONV_B09 takes the .B09 file and gets the object code out of it and converts it to decimal code.

```
      ∇CONV_B09[⎕]∇
[  0]    R ← CONV_B09 FILE ;⎕IO;I
[  1]   (FILE,'.B09') ⎕TIE ⎕IO←1            ⍝TIE THE .B09 FILE TO THE WORKSPACE.
[  2]   R← ⎕GET 1 2000                      ⍝GET ENOUGH CHARACTERS (MAY NEED TO
[  3]   ⎕UNTIE 1                            ⍝..INCREASE IF VERY LARGE FILE).
[  4]   I←¯1↑(R='T')/⍳⍴R                    ⍝FIND THE INDEX OF 'T'.
[  5]   R←(I+1)↓R                           ⍝TAKE THE OBJECT CODE FROM .B09.
[  6]   R←(⎕AV[⎕IO+13]≠R)/R                 ⍝COMPRESS OUT ALL CARRIAGE RETURNS.
[  7]   R←(((⍴R)÷2),2)⍴R                    ⍝MAKE A 2 COLUMN ARRAY OF HEX BYTES.
[  8]   R←16⊥-⎕IO-'0123456789ABCDEF'⍳⍉R     ⍝CONVERT THEM TO A DECIMAL VECTOR.

R←SYS CONV_B09 'MLDUMP'             ⍝USING THE FUNCTION 'SYS' EARLIER DEFINED.
```

The .B09 file holds some English instructions; the object code in hex follows. Just before the hex code the title "OBJECT" appears, followed by a carriage re-

turn. Because OBJECT ends in a "T", we can use an APL idiom which locates the index (position) of the last appearance of any character in a vector to find the index of the last T. Line 4 in CONV_B09 does this. Line 5 gets us the object code by taking the characters following the carriage return after this T. The object code is broken up here and there with carriage returns. Line 6 gets rid of these carriage returns, and we now have a vector of hex characters. We take these in pairs and form a two-column matrix of the pairs (line 7). Finally they are converted to decimal code by line 8. We could have broken this program up into parts and written it using the direct function definition compiler, but the first part of it can't be written this way since it is not result-returning; we wrote it all as one function, using the del editor. Executing the final program given in the above example resulted in a screen dump to printer, as expected.

We have successfully loaded in large .B09 files using this method. There doesn't seem to be any limit to the number of bytes which can be "GOT", except for work-space size. Large files, however, are prone to workspace-full limitations when you run CONV_B09. If this happens, try bringing the .B09 file in, assigning it to a variable, and converting it by hand using the steps in CONV_B09. Eliminate any unnecessary objects in the workspace at each step to maximize space available. We have successfully converted a 2400-byte .B09 file this way.

The beauty of this method of executing machine language programs in APL is that it's not necessary to restrict the APL workspace to create a safe place to store the machine language. If the "ceiling" (or end of user memory of the APL work-space) is reduced below $7FFF, other workspaces will be incompatable and you can not)LOAD them. Functions may be copied in with)COPY or)PCOPY, but this is a slow process. After converting a .B09 file with CONV_B09, the resulting vector of decimal code may be kept as a global variable in the workspace or in a seq-ential file on disk and loaded in when needed. The program is easy to edit in APL by modifying the decimal code, which is another advantage.

The reader is encouraged to load in a .B09 file within APL by using lines 0 through 3 of CONV_B09. Take a look at it and see where the word "OBJECT" (with the APL character for "capital" O) appears. The method used for extracting the object code will then be more easily understood.

QUAD POKE and PEEK are slow in APL. We wrote machine language routines to find the address of the FCB and RESET the buffers. First, we redefined SYS to allow an optional parameter list as a left argument. If the optional list is not re-quired a discard parameter can be supplied. If SYS were not redefined this way, it would be necessary to define two forms of SYS. We assembled a slightly modi-fied form of reset (the subroutine of Stan Brockman's Assembly Language program in the article on direct access printed in the last issue of the Gazette) to make use of FCB2, passed as a parameter. The .asm files for RESET and for FCB are:

```
;reset routine
        TFR     d,x       ;Transfer fcb2 from D to X. fcb2 is passed into the
        LDX     4,x       ;..routine as a parameter by QUAD SYS in APL.
        LDB     12,x
        ANDB    #$fe      ;The remainder of the routine is the same as
        STB     12,x      ;..Stan's subroutine.
        LDB     1,x
        ANDB    #$e7
```

```
        STB     1,x
        RTS

;fcb routine
        LDD     $0073   ;The address of FCB (-1) is found in hex address $73.
        ADDD    #1      ;Finds the address of FCB and puts it in the D register
        RTS             ;..where it will be passed into APL by QUAD SYS.
```

Once these are assembled and the .B09 files formed we go to APL and use CONV_B09 to get the decimal machine language for SYS. We enter this directly into READ̄.

```
      DISPLAY 'SYS'
SYS:α ⎕SYS ⎕AV[⎕IO+ω]  ⍝REDEFINED TO ALLOW OPTIONAL PARAMETER LIST (α).

⍝REDEFINE THE FOLLOWING LINES IN 'READ' FOR MACHINE LANGUAGE FCB
⍝AND RESET.

[ 0]    Z ← READ A ;FCB2;M
[ 4]   FCB2←1 SYS 220 115 195 0 1 57   ⍝OPT. PARM IS THROWN AWAY.
[ 6]   M←FCB2 SYS 31 1 174 4 230 12 196 254 231 12 230 1 196 231 231 1 57
```

The example above shows the modified lines in READ. Using this form of READ, we were able to read in a sector in about two seconds. Using the other form, from six to seven seconds were required.

DIRECT DISK ACCESS IN 6809

by Reg Beck

Part II

In Part I (last issue) we examined the structure of File Control Blocks (FCBs) and discussed what had to be done to overcome deficiencies in the Waterloo DOS/ROMS when you use U1, U2 and B-P in direct access. An mBASIC example showed how to use the buffer pointer (B-P). Now we'll examine an assembly language routine provided by Stan Brockman which looks at the first file in the disk directory; if it is a scratched file, it restores (unscratches) it as a SEQ file and then renames it to "DUMMY FILE". In the process, it demonstrates how to get data from the disk with the U1 and B-P commands and then write data back to the disk with the U2 command. To quote Stan, "There are much better ways to rename a file, but this shows all the ways of getting the data off the disk--and back." The parts of the routine are numbered for convenience when we later discuss each part.

```
;rename--Direct access example. Demonstrates the way to get the U1, U2 and B-P
;  19 Dec 1985     commands to work. For 4040 drive. To modify for an 8050,
;  Stan Brockman   change the number 18 in u1 and u2 (at end of program) to 39.
;
;include<call.macro>
openf_        EQU   $b0ae
closef_       EQU   $b0b1
initstd_      EQU   $b0ab
mount_        EQU   $b0e7
fprintf_      EQU   $b0c9
fgetchar_     EQU   $b0d8
fputchar_     EQU   $b0cf
cls           EQU   $d9a9        ;Rom routine to clear the screen.
```

```
; part 1
        JSR   initstd_
        JSR   cls                ;Clear the screen.
        CALL  mount_,#disk0      ;Initialize drive 0.
        LDU   #$8000
        LEAU  -256,u             ;Set up a buffer area into which the disk
        TFR   u,d                ;..buffer will be copied.
        STD   tmp_buf            ;Save the buffer address.
        CALL  openf_,#ch15,#UU   ;Open command channel
        STD   fcb15              ;..and save the address of the FCB.
        CALL  openf_,#ch2,#UU    ;Open channel to disk buffer (req'd by
        STD   fcb2               ;..U1/U2) and save FCB address.
        LBSR  reset              ;Clear bits 3 & 4 of access byte and bit 0
                                 ;..of status byte in channel 2FCB, just in case.
; part 2
;* * * Send U1 command to disk to read track 18, sector 1 of drive 0. * * *
;
        CALL  fprintf_,fcb15,#u1       ;This sends the U1 command but nothing
        CALL  fgetchar_,fcb2           ;..will actually get read until a char
                                       ;..is gotten from the buffer.
; part 3
;* * * Set buffer pointer and read the whole sector from buffer into SPET. * * *
;
        CALL  fprintf_,fcb15,#b_p      ;Set buffer pointer to position 0.
        CLR   -1,u                     ;Set up an iteration counter,
        LDY   tmp_buf                  ;..load addr of SPET buffer into Y,
        LOOP                           ;..and loop 'til whole block is read.
              CALL  fgetchar_,fcb2           ;Get a char,
              STB   ,y+                      ;..and store it.
              DEC   -1,u                     ;Decrement counter
        UNTIL EQ                             ;..until it is 0 again.
        LBSR  reset                    ;Reset the channel 2 FCB flags again.
; end of part 3
;* * * Set up first directory entry with new name. * * *
;
        LDX   tmp_buf
        LDB   #$81
        TST   2,x          ;Test the file-type byte.
        IF EQ              ;If it is null,
              STB   2,x          ;..store code for SEQ file there.
        ENDIF
        LEAX  5,x          ;Now point at the first name (if any)
        LDY   #nu_nam      ;..and load address of the new name.
        LOOP               ;Transfer new name to the first directory entry.
              LDB   ,y+
              QUIF EQ            ;Exit loop if end of NU_NAM has been found.
              STB   ,x+
        ENDLOOP
        LDB   #$a0         ;Load B with shifted space character.
        LOOP
              CMPB  ,x           ;Pad rest of name with #A0, as necessary.
              QUIF EQ            ;Quit if entry already has $A0 in curr. pos.
              STB   ,x+          ;This loop can bomb if previous filename was
        ENDLOOP            ;      ..16 characters long.
```

```
; part 4
;* * * Now send it all back to the buffer again, then write it to disk. * * *
;
        CALL  fprintf_,fcb15,#b_p     ;First reset the buffer pointer.
        CLR   -1,u          ;Initialize the counter
        LDY   tmp_buf       ;..and put the buffer address into Y.
        LOOP
              LDB   ,y+
              PSHS  d
              CALL  fputchar_,fcb2
              LEAS  2,s
              DEC   -1,u
; part 5
        UNTIL EQ
        CALL  fprintf_,fcb15,#u2      ;Send the U2 command.
        PSHS  d                 ;Push whatever character is in D
        CALL  fputchar_,fcb2    ;..and send it to the buffer to force the
        LEAS  2,s               ;..buffer to be written to disk.
; part 6
;* * * Finally, close files and quit. * * *
;
        CALL  closef_,fcb2
        CALL  closef_,fcb15
        CALL  mount_,#disk0     ;Shuts the drive light off.
        CLR   $32
        RTS
;-----------------------------------------------------------------------
reset   EQU   *             ;This subroutine clears the EOF bit of the status
        LDX   fcb2          ;..flag in the second part of the input buffer FCB
        LDX   4,x           ;..so that any subsequent read will be successful.
        LDB   12,x
        ANDB  #$fe
        STB   12,x
        LDB   1,x
        ANDB  #$e7          ;Clear bits 3 and 4 of the Access Mode byte.
        STB   1,x
        RTS
;-----------------------------------------------------------------------
tmp_buf RMB   2
ch2     FCC   "ieee8-2.#"       ;The "#" says "reserve me a buffer" and attach
        FCB   0                 ;..it to channel 2.
ch15    FCC   "disk8-15/0"
        FCB   0
disk0   FCC   "disk/0"          ;Used in initializing drive 0.
        FCB   0
UU      FCB   'u,0
fcb2    RMB   2
fcb15   RMB   2
b_p     FCC   "B-P 2 0%n"       ;Will set buffer pointer to position 0.
        FCB   0
u1      FCC   "U1 2 0 18 1%n"   ;To read track 18, sector 1, of drive 0
        FCB   0                 ;..into buffer attached to channel 2.
u2      FCC   "U2 2 0 18 1%n"   ;Same as u1, but write.
        FCB   0
```

```
nu_nam  FCC   "DUMMY NAME"
        FCB   0
        END
```

The file "call.macro" which follows is saved as a separate file on the disk. It first appeared in the Gazette Vol. I No. 2, p. 158 in an article by John Toebes, and is given here for those readers who do not have that issue. The macro passes parms to routines in D register and on the stack and adjusts the stack automatically. The comments on the call macro are omitted. All comments must be deleted from this macro; the name "call macr" must begin at the left margin.

```
call     macr
    pcount_  set 0
      ifnc  .,\1.
        ifnc  .,\2.
pcount_    set   2
          ifnc  .,\3.
pcount_      set   4
            ifnc  .,\4.
pcount_        set   6
              ifnc  .,\5.
pcount_          set   8
                ifnc  .,\6.
pcount_            set   10
                  ifnc  .,\7.
                    fail
                  endc
                  ldd   \6
                  pshs  d
                endc
                ldd   \5
                pshs  d
              endc
              ldd   \4
              pshs  d
            endc
            ldd   \3
            pshs  d
          endc
          ldd   \2
          pshs  d
        endc
        ldd   \1
      endc
      jsr   \0
      ifne    pcount_
        leas  pcount_,s
      endc
      endm
```

The following comments on the numbered sections of the program above come from Stan Brockman:

"1) Opening the files is fairly straightforward. Terry Peterson observed, though, that the FCB (file control block) of a file opened to file "DISK..." will be properly initialized but that because of a bug in the Waterloo IEEE IEEE routines, the FCB of a file opened as "IEEE..." will not be. Therefore, I do a re-set of the disk buffer FCB (channel 2 in my program) after I open it. The drive actually accessed will be the one last accessed, so I avoid suprises by using the MOUNT command to initialize the drive I intend to use, prior to actually opening the channels.

2) Read a block of data with U1: Study the string constant, "u1" at the bottom of the program. The parameters must be separated by spaces. The key is to make sure that the string is terminated by a CR when it is sent to the disk on channel 15. Won't work otherwise. No action will take place until a character is "got" from the buffer (never mind that the disk hasn't yet done the read you just asked for). There have been times when the character gotten isn't the first one in the block being read, so I throw it away and go on to the next step.

3) Set the buffer pointer to position 0 with B-P. "b_p" parameters are also separated by spaces and terminated with a CR. If the entire block is transferred to SPET memory, as in the demo program, the EOF bit in the FCB will be set and will need to be cleared before any other read or write operation to the channel. Further, the read will set bit 3 of the Access Mode byte of the FCB and will prevent a subsequent write (but not another read).

4) Set the pointer again prior to sending data back to the buffer. I think this step may be unnecessary if you read in the entire block in the first place, because I understand the pointer wraps around.

5) Send the U2 command, with space separations and a CR. Force the write action by putting more data into the buffer.

6) Button up the channels, shut off the drive light by MOUNTing the drive again. Regarding steps 2 and 5, I've used unsophisticated methods because I'm simply reading/writing a desk sector at a time. A database application would probably want to take advantage of the smart DOS to effect record-spanning between disk sectors, etc."

We think there should be enough information here (and in part 1) to permit mastery of the direct access filing techniques in Assembly language, mBASIC and in APL (we hope). Stan is at present hard at work on a DISK DOCTOR program. Any corrections and/or additions to this information would be very welcome.

[Ed. note: We began this series in Vol. II, No. 7, p. 197, but couldn't continue it for several issues because of the space occupied by reviews of Amiga.]

TABLE LOOKUP, SEARCHES, HASHING and ORGANIZING DATA
Part II - Hashing

In issue 7, we showed two swift ways to find any entry within sorted lists of names and numbers--which work well if you have time to sort the lists. But sometimes you don't have time. Suppose you're taking orders by phone. How do you sort the list of today's customers and add it to yesterday's? How do you respond to a second phone call changing the order? Or assume you're entering the names and scores at a track and field meet and don't know until the events begin who'll compete or in what events. Somehow, you must take data as it arrives, store it, and then retrieve any entry very quickly.

Hashing is just such a technique, quite simple in concept and just as simple to implement, once you learn where the problems lie. It is also fast and powerful.

1. Generate an index from the datum itself.
2. Use the index to put the item in a table, unless another entry already is in the table at that place.
3. If so, find another place in the table.

We show the three fundamental steps at left. In a sense, hashing is a different way to sort, which is independent of all other entries.

Once an item is stored, you may recover it by using the same algorithm which emplaced it. For example, we might write a program which assigns the first position in a matrix to any entry beginning with the letter 'a', and the 26th place to an item which begins with a 'z'. To retrieve the item, you enter the word, parse it for the first letter, and look at that position in the array.

The problem with this approach is immediately evident: where do we put the second entry which begins with "a"? We confront the problem of entry collisions, which is inherent in all hashing approaches. There are an enormous number of different ways to deal with collisions and to quickly find a happy home for any entry despite collisions. We'll later cover some of them.

It is entirely possible to generate from any unique entry, such as "Joshua Smed Bruner," a hashed index which you might think would be as unique as the name (manipulate the values of every letter in the name, for example). You may at first guess that this is the way to avoid collisions. It isn't. The number of possible entries (and the number of possible indices into a table) is infinite.

No computer we know about can store a matrix of infinite dimensions. Suppose a hashing algorithm can generate 15,000,000 unique indices, and you can store only 1500 entries; any unique key must be converted to one of 1500 not-so-unique index numbers--and you'll still have collisions. The art of hashing is in large part the art of compensating cleverly for collisions when they do occur.

The more densely a matrix is packed with entries, the oftener entries collide. The term "loading factor" is the ratio of items stuffed in a table to the capacity of the table. If a matrix with a capacity of 500 items holds 250 entries, the loading factor is 250:500, or .5. We can also state it as a "packing" or "loading" of 50%. When you hash, you are well advised not to exceed a 60 per cent loading if you want speed. Size your tables with this in mind.

Let us now look more closely at a simple hashing algorithm for use with character entries, and at ways of compensating for collisions. For simplicity, we use a starting base of 1 (all matrices begin with element 1, not 0).

* * *

A Simple Algorithm to Generate a Hashing Key

sum = ASCII ordinal of character 1 * 4 + ordinal (next-to-last character)
key = mod(sum, size_of_matrix) + 1

["key" is the index or subscript into a table or matrix]

In the algorithm above, "ASCII ordinal" means the ASCII code number for a character. "Sum" is an intermediate variable which weights the first character of an entry by four, and adds the ordinal of the next-to-last character to distinguish an entry from others with the same first letter. The index into the matrix itself is "key", which is formed modulo the size of the matrix. As an example, we use the word "star". The ASCII ordinal of "s" is 115; we get 460 when we multiply by 4. When we add 97--the ordinal of "a"--we get a final sum of 557. Our key into a small matrix of 52 elements then is: mod(557,52), or what's left over when we divide 557 by 52. The answer is 37--plus one (we have no matrix element zero). We therefore place the word "star" in matrix element 38. Two aspects of the algorithm deserve further explanation:

1. Why do we use modulo arithmetic? The ASCII ordinals of the alphabet range from 65 to 122. We can't use them directly, for there is no way to fill elements 1 through 64 of any matrix. Second, if we subtract 64, so that "A" becomes 1, we find that most-used characters ("s", "e", etc.) will form large clumps, while rare characters ("z", "x", etc.) have an equal amount of space reserved within the matrix but very few entries. We must distribute all entries as evenly as we can throughout the matrix. We do this by making "sum" so large that the modulus of it bears little if any relation to normal ASCII character values. Third, in the ASCII character set, we find six non-alphabetic characters between "Z" and "a"--and we certainly don't want to waste matrix space for them. Modulo arithmetic solves all these problems very easily.

2. Why do we multiply the ordinal of the first character by 4? First, to get a large value for "sum", and second, to avoid an indiscriminate median value for entries. The sum of the ASCII ordinals for "F" and "o", for example, is 70 + 111 or 181, but so is the value of "E" and "p". Simple addition always clusters values in a small part of a matrix. Whatever multiplier is used to make "sum" large must vary with circumstance. The multiplier of 4 is much too small for a matrix of 5200 elements, because the largest "sum" we'd ever generate would occur from

two "z's", and would total only 610. Matrix elements 611 through 5200 we could never fill. You must adjust the multiplier for the job.

Now that we have an algorithm, let's explore collisions as we try to hash similar words into a matrix.

English is full of words and names in which the 1st, 2nd...nth characters are identical (as are the first six characters of the "bright" series, at left). How do you ever hash such words quickly with minimal collisions? There are many approaches; we'll discuss some variations.

abstain,abstainer,abstains
bright,brighter,brightest

The first approach is to generate a unique index for each word (we previously showed why that fails). The second is to avoid collisions by using the last character in a word in combination with the first. Unfortunately, that is bad; a few characters (t, s, g, y, r, e, etc.) seem to predominate at the end of similar words and names. We had more success when we used the next-to-last or next-to-next-to-last with the first. Fourth, if you try to employ a large number of characters in combination, you slow the algorithm down too much; it's faster to compensate for collisions than to avoid them. We illustrate the problems below:

We made up a list of similar words with an unholy predominance of "b's" and used the hashing algorithm above on it. We show the values from the algorithm at left and print a small part of the list to show what happens. Note the values shown for the index, or "key" on those words which are much alike.

Ord*4	Last-1	Sum	Key	Word
388	105	493	26	abstain
388	101	489	22	abstainer
388	110	498	31	abstains
392	104	496	29	bright
392	101	493	26	brighter
392	115	507	40	brightest
420	101	521	2	item
420	122	542	23	itemize
420	109	529	10	items
392	101	493	26	beaker
392	97	489	22	beak
392	97	489	22	bead
392	101	493	26	borden
392	101	493	26	bored
392	101	493	26	border
408	111	519	52	fox
408	101	509	42	foxes
408	120	528	9	foxy
412	115	527	8	ghost
412	116	528	9	ghosts
412	101	513	46	ghosted
392	105	497	30	brim
392	109	501	34	brims
392	110	502	35	brimming

As you scan the list of words and the index for each, note the separation of index values for some: (e.g., "abstain", "item", "ghost") --and then look at the listings of words beginning with "b", where our entries collide badly.

When we picked the next-to-last character to resolve collisions, we failed to allow for the fact that "bored" and "borden" and similar entries will always collide. We tried many different ways to form indices without much luck.

We can choose instead to use the next-to-next-to-last character (or any other combination) but soon wonder what we'll do about short words (how do you separate "fox", "foxes" and "foxy" if you don't use the last characters?). Well, we tried the next-to-next-to-last; we tried using the values of more characters; tried various weighting schemes. None of them improved performance significantly--but all the more complex methods did significantly slow down hashing.

We decided that the solution to collisions lay not in a more sophisticated hashing algorithm, but rather in a acknowledging and compensating for collisions, as with death and taxes. The next article covers that approach.

DOUBLE HASHING AND SOME SIMPLE VARIATIONS

When a new entry hashes to the same location as an entry already in place, the obvious solution is to rehash and generate a new key or index. This is called "double hashing." We start with a very simple method. It creates a new key by adding a constant offset to the old "key" each pass throught a rehashing loop until it finds a free spot in the matrix. See method at left, below. It efficiently generates a series of unique numbers, none of which is ever duplicated, which in time will fill every element in a matrix. We show below a series generated by the algorithm for a matrix of 50 elements. Take our word for it that every number between 1 and 50 is in it. The method also generates a series which is unique for any specific key. You may load a matrix 100% full with this algorithm (if you're careful on several points we'll note below), since each series generates every element in a matrix:

key% is previously hashed key

```
while matrix$(key%) > ""
  key%=mod(key%+offset%,size%)+1
endloop
```

size% is size of matrix

Re-Hashing Keys Generated in Matrix of 50 for the Key of 25

34 43 2 11 20 29 38 47 6 15 24 33 42 1 10 19 28 37 46 5
14 23 32 41 50 9 18 27 36 45 4 13 22 31 40 49 8 17 26 35
44 3 12 21 30 39 48 7 16 25 <-- The last key generated is the original.

The algorithm above will not always generate all the elements of a matrix IF the matrix size is not a prime number. You may avoid this restriction very easily: if the size of the matrix is an even number, the offset must be even; likewise, with odd-sized matrices, the offset must be odd. The offset itself may range from 1 on up to the size of the matrix -1; sometimes larger offsets work well; sometimes they don't. If you watch these two points, you may use matrices sized to any positive value; double hashing inevitably will find a home in any unfilled matrix for any entry. Double hashing is efficient in both high-level and assembly languages.

Offset < matrix size

Offset and matrix size must be both odd or both even.
or
Matrix size must be prime number

Once you work with constant offset, you see the flaw: The series of numbers generated by key 25, when rehashed, is always the same. In the series above, the first rehashed entry goes into location 34, the second to 43, the third to 2. Always. Should you have a fourth word which keys to element 25, you must inspect and discard the first three locations before you find room for the new entry. While double hashing with constant offset obviously should delight a mathematician's heart, it doesn't delight anybody with its performance.

First Variation on the Theme: Are there simple ways around the problem above? The answer is yes, but none of the solutions is a universal panacea. We now examine the first. We use double hashing with a variable offset--which will rarely generate the same re-hashed series for any key (the series for 25, above, will almost always be different). As with constant offset hashing, this method gener-

ates all the values in a matrix; it sometimes wastes time by generating a non-unique series for a re-hashed key (a series similar to that for some other key). Even so, it's more efficient than constant offset hash in all circumstances we have tested. We define the variable offset method below:

```
pop% = ord(last character)

while matrix$(key%) > ""
  key% = mod(key%+pop%,size%)+1
endloop
```

The variable offset is defined from the last character of the word (as pop%, at left). In all other respects, this method is the same as constant offset--"pop%" merely being substituted for "offset". Remember that double hashing won't work properly if matrix size and offset are not matched (both odd or both even). If one is odd and the other even, you obtain a few (sometimes only four) good new indices, and then repeat the same new keys infinitely, as in: 43, 34, 2, 28, 43, 34, 2, 28, 43--which would seem to force us to make "pop%" either odd or even to match our matrix. Nevertheless, we can use this simple method as-is, most effectively. We'll call double hash with a differing offset "delta hashing" for want of a better name.

Because we know that any rehash after the fourth new key may begin to repeat, we abandon delta hashing after the fourth rehash and simply increment or decrement the last key until we find a home for a new entry. Although this is certainly not elegant, it is the most efficient method we have found for hashing words into matrices up to 80% loaded. The reason is not hard to find--any hashing sprinkles entries into small clumps in a matrix; when the loading factor is .8 or less and rehash fails four times, you have jumped from clump to clump, and simply sneak away from the clumps with a few decrements or increments. It works. In actual runs, the "decrement" feature was never needed in matrices loaded to 60% or less. In trials, we decremented, so we call the method "delta hash with decrement". Increment should work just as well.

```
pop% = ord (last character)
while matrix$(key%) > ""
  key% = mod(key%+pop%,size%)+1
  j = j + 1
until j => 4
while matrix$(key%) > ""
   key% = key% - 1
   if key% < 1 then key% = size%
endloop
```

Second Variation Since plain delta hashing will generate every element in a matrix (provided the matrix size and offset are both odd or both even), we revised the method to insure that "pop%" was always even to match our matrix. This (call it: delta hash with even offset) is far and away superior in heavily loaded matrices, but not nearly as good in the lightly loaded ones. The choice of method depends on the application. This method can be modified to use only odd numbered offsets if your matrix size turns out to be odd.

```
while matrix$(key%) > ""
   key%=mod(key% + pop%, size%)+1
   if fp(key%/2) then pop% = pop%+1
endloop
```

Third Variation Last, we sized matrices to prime numbers and used the plain variable offset algorithm we began with (left).

```
while matrix$(key%) > ""
   key%=mod(key% + pop%, size%)+1
endloop
```

In the table below, we compare the performance of the three variations on a matrix of 160 (it was 163 for the prime version; we adjusted the results to make results comparable) for three different load factors:

	60% Loading			80% Loading			Full Matrix		
	Prime	Decr.	Even	Prime	Decr.	Even	Prime	Decr.	Even
Passes to insert Worst Item	4	4	6	7	6	11	42	122	31
Entries Placed in 1st or 2nd Hash (Per Cent)	90%	94%	89%	78%	85%	84%	66%	71%	70%
Total Collisions (Rehashes Needed)	39	36	48	114	97	101	445	866	418

See the efficiency with which random items are stored in a 60% loaded matrix. In the best algorithm, only 6% must be hashed more than twice. Because any search algorithm will hash in the same way, you may recover 94% of the items from a matrix in no more than two hash passes.

As you can see, there is no all-around winner. Be warned that the results above depend upon the list hashed; they'll change if the distribution of words in the list changes. For the type of list we worked on, we favor delta hash with decrement for lightly to moderately loaded matrices, but would use an even delta hash if they were densely packed. Delta hashing to a matrix sized to a prime number is a pretty good bet under all conditions--if a prime number is close to the size of the matrix you need.

So far, we've discussed hashing entries into a matrix. How about searching the matrix to find or compare them?

* * *

ON SEARCHING A HASH TABLE When we found that our searches for words not in a hash table were mighty slow, we suddenly realized that any null entry in a table ends the search. The hashing algorithm stuffs an entry into the first null position it finds. You therefore must end a search for a "not there" item as soon as you encounter a null entry. It's very hard to find a null entry in a matrix 100% loaded, so don't try. You search for an entry using almost the same algorithm which emplaced it (see program, below).

We pull all the pieces together in the short demonstration program below, which employs delta hash with an even value for "pop%". We tailored the program for small matrices (20 - 60), and adjusted the value of the original key for them. Provide your own list of words as input. Try the program with matrices lightly loaded and almost full; vary the ways in which the key is generated; change it to a decrement delta hash; to a prime hash; enjoy yourself, and, above all, be sure the matrix size is even (or meets the requirements of the type of hash you employ)!

Delta_Even, a Delta Hash and Search Program for Small Matrices

Provide your own input list from disk, program or keyboard

```
input "Enter size of matrix: ", size%
input "Enter loading of matrix as decimal fraction: ", loading
option base 1 : dim matrix$(size%)     ! "loading" must be a REAL variable;
                                       ! all other variables are integers.
for i% = 1 to size% * loading          ! HASH ENTRIES, FORM MATRIX
```

```
  ...input data as "word$"
  long% = len(word$) : toggle% = 0
  sum% = (ord(word$(1:1)) * 4) + ord(word$(long%-2:long%-2))
  key% = mod(sum%,size%) + 1          ! Generate original key.
  while matrix$(key%) > ""
    if toggle% = 0                      ! Rehash if required.
      pop% = ord(word$(long%:long%))/3  ! Make smaller for small matrices.
      if fp(pop%/2) then pop% = pop% + 1
      toggle% = 1
    endif
    key% = mod(key%+pop%,size%) + 1
  endloop
  matrix$(key%) = word$               ! Enter word in open matrix element.
next i%
mat print matrix$                     ! Examine the matrix on screen.

loop                                  ! SEARCH LOOP
  input "Enter word to be searched for: ", word$
  if word$ = "Mother-in-law" then quit
  long% = len(word$) : toggle% = 0
  sum% = (ord(word$(1:1)) * 4) + ord(word$(long%-2:long%-2))
  key% = mod(sum%,size%) + 1
  while (matrix$(key%) <> word$) and (matrix$(key%) <> "")  ! Note change from
    if toggle% = 0                                           ! Entry algorithm.
      pop% = ord(word$(long%:long%))/3 : toggle% = 1
      if fp(pop%/2) then pop% = pop% + 1
    endif
    key% = mod(key%+pop%,size%) + 1
  endloop
  if matrix$(key%) = word$
    print "Entry found at element"; key%
  else
    print "Entry not found."
  endif
endloop
```

Is hashing a subject now neatly settled? No. What will happen if we delete an entry (create a null entry) and then search for a different entry? Our search algorithm will stop searching when it finds any null in its path... Unless you expect to revise a matrix, this is no problem. But if you do, you must solve it.

PASCAL : CALLS AND LOCAL PARMS

By Marvin E. Cox
4900 West 96th Street
Oak Lawn, IL 60453

Perhaps Gazette readers would like an explanation to slice through the fog of call by address, call by value, and local parameters in Pascal. Call by address is also known as call by name and is identified in a heading as 'var variable:type'. Call by address gives both input to and output from the block by means of the variable. A function name declaration is also a special call by address. There is no 'var' but the function name does permit input to and output from the block.

Call by value occurs only in a heading and is identified as 'variable: type'. Call by value permits input to the block via the variable but does not permit output from the block via that variable.

A local variable is declared as 'variable:type' in the body of the block. The local variable does not permit either input to or output from the block via that variable.

A global variable is declared in the main program as 'var variable: type'. The global variable permits communication with any block without parameters via the global variable. Any function or procedure which has no variable declarations in either its heading or body treats all variables in the main as global variables.

Local variables generally avoid side effects. Call by value requires that the programmer be alert to avoid side effects. Call by address may cause even more side effects. Global variables are usually avoided because inadvertant duplication of variable names with those in the blocks can become a disaster.

The programs which follow demonstrate the methods we have examined.

```
program call_by_address(input,output);
var number:real;
procedure square1 (var callin:real);
  begin
  writeln('A    ',callin:8:4);
  callin:=callin*callin;
  writeln('B    ',callin:8:4)
  end;
begin
number:=10.2;
square1(number);
writeln('C    ',callin:8:4)
end.
```

The printout is:

```
A    10.2000
B   104.0400
C   104.0400
```

In the procedure above, the procedure heading parameter 'callin' is preceded by the keyword 'var'. This permits transfer from main to the procedure and also permits transfer from the procedure back to main. The writeln(A   ',callin:8:4), etc., allows us to match the printout to the program listing so we can easily trace what is going on in this call by address.

If we remove the keyword 'var' from procedure square2 heading which follows, we have call by value. In procedure square2 the call by value heading permits us to transfer of the variable from main to the procedure square2 but will not permit transfer of the variable from procedure square2 back to main.

```
program call_by_value(input,output);
var number:real;
procedure square2(callin:real);
  begin
  writeln('A    ',callin:8:4);
  callin:=callin*callin;
  writeln('B    ',callin:8:4)
  end;
begin
number:=10.2;
square2(number);
writeln('C    ',number:8:4)
end.
```

The printout is:

```
A    10.2000
B   104.0400
C    10.2000
```

If we completely remove the parameters from the next procedure heading (square3) and declare the variable in the procedure declaration section, we obtain a local variable in the procedure. If we wish, we can use the same variable name in the procedure as we did in the main--but the two variables with the same name can concurrently have two different values. This effectively isolates the procedure from the main and from the other procedures which use local variables. It gives us the ability to use freeze-dried procedures without much regard to duplication of variable names.

```
program local(input,output);
var number:real;
procedure square3;
  var number:real;
  begin
  number:=6.0;
  writeln('A   ',number:8:4);
  number:=number*number;
  writeln('B   ',number:8:4)
  end;
begin
number:=10.2;
square3;
writeln('C   ',number:8:4)
end.
```

The printout is:

```
A    6.0000
B   36.0000
C   10.2000
```

Global variables are declared in main as 'var variable:type', while the procedure is lacking in header parameters and in declared variables. Global variables give the superficial appearance of being very easy to use but create pitfalls for the unwary. Global variables generally preclude the use of freeze-dried procedures in a program.

```
program global(input,output);
var number:real;
procedure square4;
  begin
  number:=6.0;
  writeln('A   ',number:8:4);
  number:=number*number;
  writeln('B   ',number:8:4)
  end;
begin
number:=10.2;
square4;
writeln('C   ',number:8:4)
end.
```

The printout is:

```
A    6.0000
B   36.0000
C   36.0000
```

As you can see, the four examples above all give different results. Each is a tool and each should be mastered.

A function in Pascal is a special kind of call_by_address procedure in which both the procedure name and a call_by_address variable in the heading are replaced by the function name. This revised heading can in turn contain call_by_address parameters, call_by_value parameters, or variables can be declared in

the declaration section. Thus, functions follow the same rules as the procedures we just reviewed. You may readily convert procedures to functions; functions can be easily converted into call_by_address procedures. As an illustration, function square5 below is the conversion of procedure square1, printed above.

```
program conversion(input,output);
var number:real;
function square5(number:real):real;
  begin
  writeln('A   ',number:8:4);
  number:=number*number;
  writeln('B   ',number:8:4);
  square5:=number
  end;
begin
number:=10.2;
number:=square5(number);
writeln('C   ',number:8:4)
end.
```

The printout is:

```
A    10.2000
B   104.0400
C   104.0400
```

Note that the printout is identical to that for proc square1.

[Ed. Note: A lot of you have run into keyboard bounce. We've updated this article on how to cure bounce with some new and better ways.]

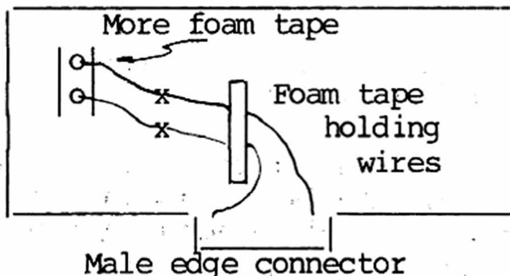
BABIES, NOT KEYBOARDS, SHOULD BOUNCE

When your 'a' key starts to print 'aa' or your keypad doesn't SHIFT when you press SHIFT, you are faced with dirt in the keyboard enclosure. Being over 300 miles from our dealer, we gave him a call when, after 9 months, the keyboard on one SPET became non compis mentis. He advised us to take it apart and clean it. We did. It worked. Since that first time, we've learned a lot about how to avoid keyboard 'repeat' problems. We cover two routes below: (1) surgery, which cures the patient for quite a while, and (2) palliation, which postpones it.

Option 1: Surgery. If you can handle simple tools, surgery is easy but takes about two hours. Disconnect power and open SPET. You'll see a host of Phillips-head screws holding the keyboard sheet-metal enclosure. Disconnect the aft cable connector gently and then remove the screws, the enclosure, and the keyboard itself. Clean the enclosure and under the keys. An air hose at low pressure works well for the latter. Then buy some thin foam insulating tape (see below).

Get a good, small screwdriver. Turn the keyboard over. You will see a multitude of tiny screws, which hold the epoxy-glass keyboard bottom to the keyboard. Remove them (tweezers come in handy). Then HALT. Below, left, is what you'll see:

View of Keyboard, Upside Down



As you might expect, the two wires which run between the edge connector and the interior of the keyboard are too short to let you remove the bottom of the keyboard. Cut at 'x' or unsolder them at 'o'. They're color-coded to prevent reconnection problems. Then gently remove the bottom board, and behold the small, dead spiders, remains of lunches, and similar debris. We cleaned the outside and inside with clean cotton swabs lightly soaked with rubbing alcohol. Suggest you not rub long or hard on the gold printed capaci-

tors and circuits on the interior of the board. When the board is clean, you've solved half the problem.

The other half lies in the glaze of dirt on the black, graphite-impregnated tips on the keyboard key contacts. Each key tube ends in a gray rubber diaphragm; on the bottom of each diaphragm is the black, circular contact you must clean. The glaze does not come off easily. You can clean with swabs and alcohol, but you'll be bouncing again in two weeks. Best method: take diaphragm out with tweezers; slip it on the eraser-end of a pencil, and twirl the black contact on a piece of flat, clean aluminum oxide sandpaper (150 grit is fine). Don't take off more of the black contact than is necessary to remove the glaze. Note how jet-black the sanded contacts are. Keep the pencil vertical (keep the black contact FLAT). Be sure not to leave a whisper of glaze. The black dust from sanding, and grit from the sandpaper, is deadly. Keep it away from the keyboard! Clean the diaphragm thoroughly before you insert it in the keyboard again.

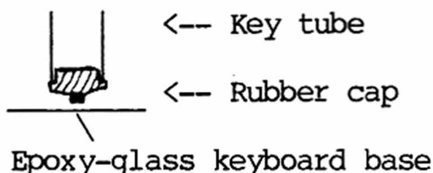
With everything squeaky-clean, put the bottom board back on the keyboard; thread the two wires through the holes marked 'o' on the diagram above. If you can solder, solder the wires together again (rosin flux only--NO zinc chloride, unless you like hydrochloric acid in your connections!). If you cannot do a solder job, use the thinnest crimp-on connectors you can find (no thicker than the foam tape). Now put those tiny screws back in to hold the bottom board. This done, replace the missing foam tape; insulate the spot you cut the wires with a couple of flat layers of vinyl electrical tape. You're ready to put the assembly back in SPET.

One last tip: shim the FRONT keys of the board away from the plastic case with a few layers of thin cardboard before you tighten the screws. With SPET open, the keyboard wants to fall down hill; it crams the top row of keys so tightly against the case that they won't move--unless you shim to space them properly. When all screws are tight, remove the shims.

We've cleaned and reconditioned keyboards twice a year since we bought our machines, with utter success. Both SPETs operate perfectly today. Commodore has not reviewed the procedure; you proceed at your own risk.

Option 2: Postponing Surgery. With SPET off, slip a thin-bladed screwdriver under the offending keycap; pry up gently until key and return spring come off. Insert a clean Q-tip in the key well, and gently rotate the top of the Q-tip in a circle, pressing gently down. Then reverse the rotation. You thus scrub the rubber key contact against the printed capacitor on the base board. This usually cleans both sufficiently to get rid of 'repeats'--for a while, but it is not a permanent cure. You probably will have to do it again in a month or so.

Vertical Cross-Section



You have yet another palliative, which takes a little longer but also is longer-lasting. After the scrubbing above, get a piece of pipe-cleaner [the fuzzy l'il worm with which you clean tobacco (not plumbing!) pipes]. Cut a piece just long enough to reach from the bottom of the well to the keycap when installed. Wrap both top and bottom of this with cotton, as though it were a Q-tip. Stuff it in the well. You'll have to try a few times to get the length just right. Keep a sample for next time! Then stick the keycap back on. The additional pressure ex-

HRT Super-Res Graphics Board

from High Res Technologies

An internally-mounted, high-resolution graphics card for PETs and SuperPETs

Review by Tom Stiff

In my work, I accumulate a lot of data; data that must eventually be analysed. As anyone who has had to handle large amounts of data will attest, the best way to begin to understand the meaning of large blocks of related data is to display it graphically. The old adage "a picture is worth a thousand words" never held more truth than in the field of data analysis.

In a single night's observing at the York University Observatory, I can easily accumulate more than a megabyte of data in the form of digitized one dimensional stellar spectra, or a few dozen megabytes in the form of two dimensional digitized astronomical images. Producing a hard-copy of these images is very expensive and time consuming. A fast graphics terminal seemed to be the best solution, and I began a search for a PET or SuperPET graphics package.

Months of letter writing, phoning and searching through back issues of every Commodore-related magazine failed to produce any satisfactory results. In the Spring of 1984, there were three manufacturers of hi-res boards, all for 65xx based machines (designed for PETs, but not for SuperPETs); all were priced over 600 dollars (US); and none offered a screen resolution any better than the C-64.

Each system also had serious design flaws that made them unsuitable. One used up a lot of the PET's memory, another required major hardware modifications and yet another limited the PET's capabilities by redefining some of the PET's BASIC keywords to incorporate graphics commands.

At the 1984 Annual TPUG meeting, I discovered High Res Technology's booth, and described my graphics needs to Dan Deconinck. He seemed optimistic about designing a suitable graphics board. That summer, Dan contacted Avygdor Moise (of OS-9 fame, and also author of **PETCOM**) at York University, for details about the 6809 side of the SuperPET. Since my graphics needs were relatively modest, Dan also asked Avy for additional ideas and suggestions. Gradually, a prototype graphics card began to emerge.

During the next year, two prototype boards were produced. Each was demonstrated at a meeting of the SuperPET User's Group, and each time ideas for enhancements were solicited. Ideas thrashed around during these meetings led to further board revisions and improvements. In the early spring of 1985, HRT felt that their card was ready to 'field test'. Accordingly, they installed their graphics card in my SuperPET.

The graphics card was fully transparent and did not interfere with any PET functions, nor did the card use any of the PET's memory. In other words, with the card installed, I was totally unaware of its existence. I could tack graphics sub-routines onto any existing program and the program would run perfectly!

I have been using an early version of an HRT graphics card for about ten months now, and I am extremely pleased with its operation. I use the graphics capabilities of my SuperPET to analyse stellar spectra. A typical spectrum (called a frame) is 500 pixels in length. The spectra are taken with a silicon-intensified television videcon (a sort of fancy digital TV system). All of the observatory's instrumentation, by the way, is controlled by an ordinary PET 2001.

The data are written onto a floppy disk for temporary storage, then handed over to a VAX 8600 for large-scale 'number-crunching'. I then download the reduced data from the VAX to my SuperPET, for graphics display. The graphics program I have written is entirely in BASIC, and has been compiled, using **PETSpeed**.

It takes about thirty seconds to create a full screen image consisting of over 600 line segments. Most of this time is taken by the SuperPET, to execute an auto-scaling subroutine to a VAX 780 system using a VT100 terminal and operating with normal daytime user-loading. To generate an equivalent hard copy graphics display takes about twenty minutes, using an H-P plotter! Furthermore, the HRT card allows me to overlay an infinite number of frames for comparison if I need to do so, or to simply display them, one at a time. The images can be scrolled off the screen, if I wish to plot more data; and instantly scrolled back, if I want to view them again.

With the birth of Super-OS/9, additional enhancements were made last summer to increase the graphics board's capabilities, and to further increase the screen resolution. Screen resolution is 700 pixels on 80 column PETs, and 640 by 200 on 40 column PETs. But — and this is a nice feature — the total resolution is 1024 by 512 pixels, and the screen can be scrolled in all directions.

The graphics card is easily installed into any PET or SuperPET. It requires no external power supply, no soldering, and no other hardware modifications. The board simply plugs into the main board's 6502 slot, and the 6502 chip is moved to the graphics board. It works equally well on either the 6502 side or 6809 side of the SuperPET and — as if that weren't enough — it is perfectly compatible with OS-9. The icing on the cake is that OS-9 users can also use the graphics card's memory as a 64K RAM disk.

The main problem with this card is that software is scarce. You will have to write your own — in machine language, if you want it to be fast. I have a feeling that this will not be a problem for long, since there are several users that I personally know of (and probably many others) that are already developing graphics utilities that will be placed in the public domain, via TPUG.

I have also heard rumours that PH.D. Associates will be marketing a version of **PETCOM** which will support VT100 and maybe VT240 graphics with HRT's graphics board. Perhaps this upgrade will also be available to registered **PETCOM** users for a modest fee. (Are you listening PH.D. Associates?)

This card is the finest and best designed piece of graphics hardware on the market today for any microcomputer — and at any price. This little card has many excellent features and, in some areas, it out-does the illustrious Amiga. For example, I use the graphics card to produce a 'plotting window' on the top two thirds of the screen while I simultaneously use the bottom third of the screen to edit programs, as well as to display numerical results as they are being calculated. In fact, the normal text screen can overlay the graphics screen. I can choose to erase either the text or graphics, or both.

If you want to breathe new life into your old PET and re-kindle some of the enthusiasm you had the day you first lifted it lovingly out of its styrofoam cradle, this addition might be just what you're looking for.

The HRT Super-Res Graphics Board*, from High Res Technologies, 16 EngishIvy Way, Toronto M2H 3M4. Price $200.00 (US), $225 (Cdn).* □

erted at keypress by this mechanical gimmick may keep you going for as long as a year. Be sure to use the cotton--or the wire in the pipe cleaner may punch a hole in the rubber contact. Obviously, major surgery takes less time than going this route for all keys. But--usually, just a key or two act up. If you can get them under control, you may postpone a major surgery indefinitely.

HIGH RESOLUTION GRAPHICS BOARD FOR SUPERPET The January/February issue of TPUG magazine holds a first-rate article by Tom Stiff on an internally-mounted high-resolution board for graphics which works with both PETs and SuperPETs. We were so glad to see it that we wrote the Editor, TPUG magazine and asked for permission to reprint it, which TPUG gladly gave. That article is reprinted in its original format on the facing page. Thank you, Tom and TPUG!

(Article on facing page is copyrighted, 1986, by the Toronto Pet Users Group and reprinted by permission of the copyright owners.)

BETTER LATE THAN NEVER!! A Starter Disk in APL We'd have given an arm and leg about four years ago for a disk our busy-as-a-bee Associate Editor Reg Beck just sent in, which is a tutorial on APL, in APL. We tried to pry such a disk for the beginning APLer out of Barry Bogart, Steve Zeller, and a gaggle more of APL old hands, but it never quite got together. Reg has collected that work and has added a lot of his own.

The result is one helluva lot less bafflement and confusion when you first dive into the language. It's available in either 4040 or 8050 format for $6 from ISPUG at the address below. Ask for the "Beginning APL" disk if you want one.

If you order, please state your disk format!

Prices, back copies, Vol. I (Postpaid), $ U.S. : Vol. I, No. 1 **not** available.

No. 2: $1.25	No. 5: $1.25	No. 8: $2.50	No. 11: $3.50	No. 14: $3.75
No. 3: $1.25	No. 6: $3.75	No. 9: $2.75	No. 12: $3.50	No. 15: $3.75
No. 4: $1.25	No. 7: $2.50	No. 10:$2.50	No. 13: $3.75	Set: $36.00

-----------------------------Volume II-------------------------------------

Numbers 1 thru 10: $3.75 each.

Send check to the Editor, PO Box 411, Hatteras, N.C. 27943. Add 30% to prices above for additional postage if outside North America. Make checks to ISPUG.

===

DUES IN U.S. $$ DOLLARS U.S. $$ U.S. $$ DOLLARS U.S. $$ U.S. DOLLARS $$

APPLICATION FOR MEMBERSHIP, INTERNATIONAL SUPERPET USERS' GROUP

(A non-profit organization of SuperPET Users)

[] Check if you're renewing; clip and mail this form with address label on the reverse side. If you send the label, don't fill in the form below.

Name:______________________________ Disk Drive: _________ Printer:____________

Address:____

Street, PO Box City or Town State/Province/Country Postal ID#

For **Canada** and the U.S.: Enclose Annual Dues of $15:00 (U.S.) by check payable to ISPUG in U.S. Dollars. DUES ELSEWHERE: $25 U.S. Mail to: ISPUG, PO Box 411, Hatteras, N.C. 27943, USA. **SCHOOLS!: send check with Purchase Order.** We do not voucher or send bills.

This journal is published by the **International SuperPET Users Group** (ISPUG), a non-profit association; purpose, interchange of useful data. Offices at PO Box 411, Hatteras, N.C. 27943. Please mail all inquiries, manuscripts, and applications for membership to Dick Barnes, Editor, PO Box 411, Hatteras, N.C. 27943. SuperPET is a trademark of Commodore Business Machines, Inc.; WordPro, that of Professional Software, Inc. Contents of this issue copyrighted by ISPUG, 1986, except as otherwise shown; excerpts may be reprinted for review or information if the source is quoted. TPUG and members of ISPUG may copy any material. Send appropriate postpaid reply envelopes with inquiries and submissions. Canadians: enclose Canadian dimes or quarters for postage. The Gazette comes with membership in ISPUG.

ASSOCIATE EDITORS

Terry Peterson, 8628 Edgehill Court, El Cerrito, California 94530
Gary L. Ratliff, Sr., 215 Pemberton Drive, Pearl, Mississippi 39208
Stanley Brockman, 11715 West 33rd Place, Wheat Ridge, Colorado 80033
Loch H. Rose, 102 Fresh Pond Parkway, Cambridge, Massachusetts 02138
Reginald Beck, Box 16, Glen Drive, Fox Mountain, RR#2, B.C., Canada V2G 2P2
John D. Frost, 7722 Fauntleroy Way, S.W., Seattle, Washington 98136

Table of Contents, Issue 10, Volume II

Schematics--Last Chance..........273
Commodore Far from Bankrupt......274
3-Board Bug in OS9...............275
Visicalc Instructions............276
Better Ways to Exit Procedures...282
Direct Disk Access, Part II......287
Double Hashing...................294
Curing Keyboard Bounce...........300
Starter Disk in APL..............302
Gazette Will Cease Publication......273
Indenting Printer Output............274
National Repair(?) Center...........275
Bug in Bedit 2......................277
APL Express.........................282
Hashing.............................291
Pascal : Calls & Local Parms........297
High Resolution Graphics Board.....301A

SuperPET Gazette
PO Box 411
Hatteras, N.C. 27943
U.S.A.

First Class Mail
in U.S. and Canada.
Air Mail Overseas.