SuperPET Gazette

The recent drop in the price of SuperPET ($995 at mail-order houses) has caused some owners to panic. (Fire Sale! Selling out inventory! The end of all production!). Nonsense. Did anyone who bought a Commodore 64 panic when prices dropped to less than $200? Commodore obviously is employing low prices to smash its competitors--and has smashed them (Apple, caught between IBM and Commodore, has closed a plant and laid off staff at its main operation; VICTOR is about out of business; Texas Instruments has quit on personal computers; Atari is in deep trouble, and Coleco has the same problem Eve had: Adam). Commodore continues to value school installations; these have first priority on any machine. SuperPET, at $995, is far cheaper than its rivals, Apple and IBM PC. Commodore well knows that those who learn on a Commodore machine will favor that machine (or brand) after they leave school, and wants to capture the next generation of customers.

Walt Kutz, U.S. Manager of Business Machines (he was U.S. manager for SuperPET) called a few weeks back and assured us that SuperPET would remain in production. Those who see in the recent drop in price of SuperPET the impending doom of the machine might well ask themselves: would I, if managing Commodore, give schools a choice of 8032, VIC or the 64, and abandon universities to Apple and IBM?

JANUARY MEMBER, YOUR MEMBERSHIP EXPIRES WITH THIS ISSUE!

You won't get a bill or another notice. If the address label, this issue, reads e:1-nn-83 or less (nn is any day), this is your last issue. The 'e:' stands for 'entry date'--not for expiration. Any date ending in -82 means you are no longer a member of ISPUG. So please send in yearly dues of $15 U.S. for memberships in North America, or $25.00 if you live elsewhere. Enclose the address label or a copy and mark the form 'renew'. Send label and check to Secretary, ISPUG, made out to ISPUG, at 4782 Boston Post Road, Pelham, N.Y. 10803, U.S.A. Canadians: We found a bank which does not charge to process Canadian checks (to hell with the one that does!). So send anything negotiable but cash, made out to ISPUG.

On Commodore's support of SuperPET: Starting with the February issue (No. 28) of Commodore Microcomputer, Commodore's own magazine, there'll be a regular column on SuperPET--written by you know who. Diane LeBold, who edits Commodore, said she hadn't received material on SuperPET. When we offered to supply some, she accepted the offer. So there. Any of you who are interested in contributing to the column (or writing one), get in touch with the Editor at PO Box 411, Hatteras, N.C. 27943. We'll send you Commodore's editorial requirements/schedule, and try to coordinate what we've written with what you plan. We progress.

YES, WE HAVE VERSION 1.1 SOFTWARE! Through the diligent efforts of our good Secretary, Paul Skipski, we can at last lay hands on V1.1 SuperPET software for any ISPUG members who want it. We must, however, order in a batch; we can't dribble in an order now and then. Prices include the COBOL manual (sorry, we can't just get update sheets; you get the the whole package or nothing). For 8050 format, price is $29.95 plus $3.00 shipping and handling. For 4040, (which requires three disks), price is $6.00 higher. See totals at left. The package includes update sheets for ALL the manuals. To help you out, we'll replace the old microBASIC on the issue disk with patched microBASIC (debugged). There are a number of major improvements in V1.1, and we strongly urge you get it if you're still running V1.0. Almost all Gazette programs are written in version 1.1. If you want V1.1, send a check to the Editor NOW so we can put in a consolidated order. We'll need about a month to get the material and ship it.

8050: $32.95 U.S.
4040 $38.95 U.S.

(The 4040 price is based on the fact we can't get language disks in 4040, and so must convert the 8050s shipped to 4040 and supply the added disks). Order V1.1 now or forever hold your peace. Make checks to ISPUG. Send order to the Editor, SuperPET Gazette, PO Box 411, Hatteras, N.C. 27943. State format.

◇◇

ONCE OVER LIGHTLY
Miscellaneous Notes

We haven't used every brand of disk on the market, but have found two that are extremely good: Maxell and 3M; In two years, we haven't lost the first disk in either brand, employing two computers week in, week out. A box of Elephants died within a year. Want disks? Our Secretary can get some brands for members at very low prices, and supplies other stuff, hardware and software (some for the 64) at equally low rates. Write him at 4782 Boston Post Road, Pelham, N.Y. 10803 for a price list. Sample: Maxells (DD,SS) go for $27.00 per box of 10. Beat that. Add N.Y. sales tax if you live in N.Y. He also can get PAPERCLIP for you (SPET version) for $150. He has SPET manuals. Add 3% shipping and handling, all orders.

MODEM The Anchor Mark I RS-232 modem, which runs at 300 baud and sells for a modest $99.95, does a nice job with SuperPET. No fancy autodial, but after 1984 begins, dialing long distance on Ma Bell will be an exercise in patience.

SORRY! Gary Ratliff commented that Waterloo's mainframe BASIC doesn't let you pass parameters when CHAINing (letter supplment to issue nine), Waterloo sends a note: "Full-blown Waterloo BASIC for IBM 360s and 370s does allow the passing of single values of matrices to a CHAINed program. These parameters are referenced in the CHAINed program with a USE statement." We stand corrected.

NO WORDPRO 6 There will not be a WordPro 6 for SuperPET, according to Ron Patrick of Professional Software. He called and spoke for Harold Dickerman, Product Manager. The rumor is dead. Oh, well. See PAPERCLIP review, this issue.

UD11 DOES WORK Loren Felten dropped a note and said he has a WordPro ROM installed in UD11 on a two-board, two-switch SuperPET, and WordPro works fine. He checked and said POKE 61438,0 does not turn it off (that POKE turns off any chip in U45, the $9000 socket). We prefer U46, on the top board, for $A000 ROMS, as you don't have to remove the top board to get to U46. U46 works well. We have a PAPERCLIP chip there; others report that WordPro is okay in that socket.

IMPORTANT: Apparently a lot of new readers (and some old hands) never noticed the patch for microBASIC V1.1 we published in Vol. 1, p. 38. Without the patch, mBASIC 1.1 will print a carriage return after the 79th character on a line. The patch was published (V.1, No. 6), and is on ISPUG disk one as 'patch 2'. Use it!

SORRY ABOUT THAT DEPT. Last issue we published TAB, an ML program to set tabs from menu, and it works fine, but you probably noticed it makes the menu flicker twice after it runs. We found out why: library routine GETCHAR_ thirsts for CR's and won't give up until it gets one. All manner of mad things happen in ML programs if you do not feed the damned beast its CR. So, stuff the loop at left into TAB.ASM, the assembler file, right after jsr tabset_. It not only gets the CR but also gobbles up any excess characters you may have entered. Beware of GETCHAR_ in any proprogram if you leave the CR 'ungot'.

```
loop
  jsr getchar_
  cmpb #$0d
until eq
```

There's a better way to get a single character: use KYPUTB_ from the library, at $DD82. This 'gets' a character from the keyboard without

a CR (though it has some tricks of its own: see UDUMP, this issue). In TAB, you should delete the: jsr getchar_ and substitute for it the loop shown at left. KYPUTB_, as with any 'get', is happy to get a null, so you must loop until you see a character. You must stuff: kyputb_ equ $dd82 into your .asm file, right after service_ equ $32, so SPET knows where to look. Waterloo didn't document KYPUTB_, but Gary Ratliff did, on page 83 of issue 7. The character 'got' is found in the B register. Reserve GETCHAR_ for multi-character strings or to give users a chance to correct input errors.

```
loop
  jsr kyputb_
  cmpb  #0
until ne
```

Last, we said TAB wouldn't interfere with APL or COBOL if you overloaded them in Bank 15 after using TAB. Well, that's true--if you make origin $9000. We didn't; we shoved TAB way up to $9f00, and APL didn't know where APL quit and the leftover shard of TAB started. Origin should be $9000 for TAB or any other ML program in Bank 15. That low, it is surely overwritten by APL and COBOL. In other languages/facilities, you needn't change origin to $9000.

◇◇

USING MACROS IN DEVELOPMENT PROGRAMS

Part II : by John A. Toebes, VIII
145 C Jones Franklin
Raleigh, N.C. 27606

Last issue, I wrote about straight-line macros and those to which you can pass parameters as operands. They are quite simple and easy to use. This issue, we look at complex macros, and end with a general macro which will clean up the stack for you. Complex macros are the most advanced but most useful; you can do almost anything, including some high-level language constructs.

One I find most useful is CALL.MACRO; it automatically loads parameters, calls a a routine, and then cleans up the stack. It must be able to accept any number of parameters (a subroutine may have a large number of them) and then determine how much space to take from the stack. This last is a problem--the macro processor doesn't know how many parms* there are. We therefore test each parm to see if it's null; when we find the null, and if we arrange to count parms, we've solved the problem. So, let's first look at finding the null parameter. (* I use the abbreviation 'parm' for 'parameter'.)

Two conditional Assembler directives are most useful: IFC (IF Comparable), and IFNC (IF Not Comparable). (The 'comparable' means <u>exactly</u> alike. For fuller details, see p. 138, Development manual.) Both IFC and IFNC take two operands and compare them. If they are identical, the code following IFC is executed; if they are <u>not</u> identical, code following IFNC is executed. We simply test each parameter against a null until we find a match. Before I pass to an example, remember that parms are passed to macros to match a psuedo-variable, identified in the macro as '\0, \1...\n'. The first parm is assigned to psuedo-variable \0, the second to psuedo-variable \1, etc. While it might seem simple to test the parms against a null, there is a pitfall--the comma, which is a data separator.

If, for example, we pass a single parm '5,X' to a macro as psuedo-variable \1, and if we test to see if the it is the string TEST, our macro code will be that at left, top line. In operation, our code will compare TEST with 5, because 5 is followed by a comma, which excludes the following 'X', even though it is part of parm \1. We manage to sneak by this comparison through luck, not design. But now suppose we pass the parm ',S' to psuedo-variable \1 and we want to compare it to a null. We are in trouble. Our code in the macro again is

```
IFC   TEST,\1
 is seen as
IFC   TEST,5,X
```

shown at left, with the operating comparison below. We compare a prefix null to a second null, between the two commas--but we wanted to compare a null with ',S'! The solution is relatively easy. We stuff in some periods, and compare them in place of nulls, as shown in the third set of examples. (First, we'll compare a leading null to the parm ',S', and then we'll test nulls.)

```
IFC      ,\1
 is seen as
IFC      ,,S
```

```
IFC      .,\1.
 is seen as
IFC      .,,S.
```

Note the prefix period, and the period which follows the psuedo-variable \1. This time, our comparison is valid. We compare the leading period with a null between the two commas. The two are not comparable, so IFC fails as it should. Now, suppose we truly have a null parameter for psuedo-variable \1, instead of ',S'--after all, that was the problem we started with. And lo! Though we compare two periods, the comparison is valid, and we pass the IFC test. Note that the

```
IFC      .,\1.
 is seen as:
IFC      .,.
     (null,null)
```

parameter which caused all this trouble is very common:',S'--which means the top value on the stack. We had to find a way around the preceding comma, and the period is one way. Only if we pass a parm '.,'--which is highly unlikely--will the period fail. If you must compare such a parm, you can change the comparison statement to any character other than a period (if you don't otherwise use that alternate character).

Before you proceed, please look at CALL MACRO at the end of this article. I will refer to it many times from here on. The numbers in {} are line numbers, and not part of the macro. As written, it handles up to six parameters (the maximum you may pass to any system routine). If you need more, expand it. Here are some examples of how to use it; in them, your code is in CAPITALS and bold face, while lower case and '+' show macro-generated code. Note how CALL MACRO first sets the parameter count (pcount_) to zero {2}, so we can keep track of how many bytes we must release from the stack after the call. In the example left, we want a new line and pass no parameters. IFNC {3} fails because we compare two nulls, so we skip to its ENDC {33} with pcount_ still zero. The macro generates a JSR at line {34}, and pcount_ is checked at {35}. Since pcount_ is equal to zero (ifne), no LEAS instruction is generated to adjust the stack.

```
  CALL   PUTNL_
+ jsr    putnl_
```

Let's try again, with a call to PUTCHAR_ to print a space. Again, pcount_ is set to zero; the first IFNC test is passed (we pass a parm); but note the second IFNC fails. We drop down to its corresponding ENDC {31}, and emit the LDD \1 found there; it loads the #' ' that we passed (all parms to system routines are passed in the D register); then emits the JSR at {34}, and as before does not generate the LEAS instruction because pcount_ is still zero. (If you ask why passing a parameter doesn't increment pcount_, note that one parameter is passed in the D register, not from the stack. Only when we pass two parameters or more is the stack involved.)

```
  CALL   PUTCHAR_,#' '
+ ldd    #' '
+ jsr    putchar_
```

In the next call, the first two IFNC tests are passed, and pcount_ this time is incremented. Since we passed only two parms, the third IFNC fails {6}; we drop to its ENDC {28}, emit the LDD {29}, the PSHS D {30} to put P2 on stack; then the LDD for P1 at {32}, and finally the JSR at {34}. This time, the test of pcount_ shows value, and so we emit the LEAS 2,S to adj-

```
                ; P1       P2
  CALL OPENF_,#FILENAME,#FILEMODE
+ pcount_ set  2
+ ldd     #filemode  ;P2
+ pshs    D
```

```
+ ldd     #filename  ;P1
+ jsr     openf_
+ leas    2,s
```

just the stack. Note we CALL the parameters in their natural order, P1 and P2, but the macro gets the nasty chore of putting the parameters on the stack in inverse order, and of putting P1 into the D register. We begin to see the drudgery we can avoid with CALL MACRO. The next example shows it even more clearly.

Here, we have one more parm, so pcount_ is set twice, in {5} and {7}; since only the last setting counts, we are O.K. Also note there is no limit to the complexity of the parameters you pass; the macro spits them out with an LDD prefix. One warning: if any of the parms reference things already on the stack, remember that the stack is adjusted by the PSHS in the macro, so you'll have to give the stack offsets accordingly. Last, a note: if you've forgotten, the parentheses left, above, let us pass commas or even semicolons in a parameter, as we do in P1 and P3, above. Note the amount of code which the macro will write for you from one line in your program.

```
                ; P1      P2        P3
   CALL FPRINTF_,([3,X]),#STRING,(TEMP+80,Y)
+ pcount__   set 2
+ pcount__   set 4
+    ldd     temp+80,y
+    pshs    d
+    ldd     #string
+    pshs    d
+    ldd     [3,x]
+    jsr     fprintf_
+    leas    4,s
```

CALL MACRO is designed, obviously, to handle calls to SuperPET system routines. It may be shortened if you usually handle fewer than six parms, with an increase in speed. This macro should be put on disk, and brought into play with a statement at the beginning of your code: ;include <call macro>

```
{1}       call     macr           ; Call routine, parm 1, parm 2...parm n.
{2}       pcount__ set  0         ; Set stack space to release to zero.
{3}       ifnc  .,\1.             ; Test for first parameter.
{4}         ifnc  .,\2.           ; Test for second parameter.
{5}  pcount__ set   2             ; Two bytes stack space for second parm.
{6}           ifnc  .,\3.         ; Test for third parm.
{7}  pcount__   set   4           ; Four bytes stack space for 2nd and 3rd parm.
{8}             ifnc  .,\4.       ; Test for fourth parm.
{9}  pcount__     set   6         ; Six bytes stack space for parms 2 thru 4.
{10}              ifnc  .,\5.     ; Test for fifth parm.
{11} pcount__       set   8       ; Eight bytes stack space for parms 2 thru 5.
{12}                ifnc  .,\6.   ; Test for sixth parm.
{13} pcount__         set   10    ; Ten bytes stack space for parms 2 thru 6.
{14}                  ifnc   .,\7.; See if given too many parms. Six is maximum.
{15}                    fail      ; Yes, warn of too many parms.
{16}                  endc
{17}                  ldd   \6    ; Load sixth parameter,
(18)                  pshs  d     ; and push it onto the stack.
{19}                endc
{20}                ldd   \5      ; Load fifth parameter,
{21}                pshs  d       ; and push onto stack.
{22}              endc
{23}              ldd   \4        ; Load fourth parameter, etc.
{24}              pshs  d
{25}            endc
{26}            ldd   \3          ; Load third parameter, etc.
{27}            pshs  d
{28}          endc
```

```
          ldd   \2             ; Load second parameter, etc.
          pshs  d
        endc
        ldd   \1               ; Load first parameter in D register.
      endc
      jsr   \0                 ; Call the system routine.
      ifne     pcount__        ; Any stack space to release?
        leas  pcount__,s       ; Yes, remove it.
      endc
      endm
```

[Ed. When you remove the line numbers above, be sure to put 'call macr'flush on the left margin of the screen. And don't leave in the comments! If you do, IFNC fails; the macro tries to load the stack with five parms no matter how few you assign, since IFNC sees comments as part of the psuedo-variable on that line. Second, because macros slow down assembly, you may want to shorten CALL.MACRO to accept from 3 to 5 parms instead of 6. Last, you can remove the 'fail' IFNC and ENDC if you don't want to horse around with a 'too many parms' warning.]

◇◇

A NOTE ON HANDLING ASCII CODES IN APL

In issue 9, we noted the problem of handling ASCII codes with []IO set to 1 in APL. Too late for issue 10, we received a note from Dr. Wilson at Waterloo: "This letter concerns the difficulty people have when using APL in remembering to take the index origin into account in producing characters like NULL and ESC by indexing ⎕AV. For instance, the NULL character is ⎕AV[1]...with ⎕IO set to 1..., but it is ⎕AV[0] in origin 0. The BASIC function chr$( ) is not origin... dependent.

"The tradeoff is that BASIC requires a second built-in function, asc( ), to do the inverse translation of characters back to numbers, while APL doesn't need anything but the variable ⎕AV. If you prefer the two-function approach, just include in your workspace the functions CHR and ASC which I've listed below. They behave like chr$( ) and asc( ).... The results are origin-independent."

```
        ∇C ← CHR N
[  1]   C ← ⎕AV[ ⎕IO + N ] ∇
        ⎕IO←1
        CHR 97 98 99
ABC
        ⎕IO←0
        CHR 97 98 99
ABC
```

```
        ∇ N ← ASC C
[  1]   N ← (-⎕IO) + ⎕AV⍳C ∇
        ⎕IO←1
        ASC 'ABC'
97 98 99
        ⎕IO←0
        ASC 'ABC'
97 98 99
```

◇◇

A SUPERPET BULLETIN-BOARD PROGRAM BY PAUL MATZKE

Paul Matzke, PO Box 574, Madison, Wisconsin 53701, has written a program in microBASIC which allows any SuperPETter to operate a Bulletin Board as a SYSOP (System Operator). It requires a Hayes Smartmodem for autoanswer, logs all users, incorporates passwords, delivers mail, and performs the other functions which Punter boards perform in 6502. Paul says he's had it running in Madison for some time, and that almost any computer can be on the other end--from Radio Shack CoCo's to PC's (and, of course, SuperPETs and PETs). We have a report that it's the best BB in Madison, and runs well at 300 baud.

A disk with instructions, prepared by Paul, is available from him at the address above, 4040 format. Send a disk in a good mailer plus $1 U.S. (or Canadian coin)

for postage, or send $6.00 and he'll mail a disk to you. For 8050 format, send $6 U.S. to the Editor, SuperPET Gazette, PO Box 411, Hatteras, N.C. 27943, and we'll send the disk. More software, praise be. Thanks, Paul.

nnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnn

(*C*) 1984 ◇◇◇ *THE APL EXCHANGE* ◇◇◇ *STEVE ZELLER*

uuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuu

Much of the last issue was devoted to telecommunications tools. Among these were a terminal package, developed by Waterloo, that functions in APL mode. In this article, APL tools are presented that will help you communicate with other computers from the APL workspace without a terminal package. We will see that while our ability to function as a terminal is impaired, there are other important advantages to being able to communicate from the APL workspace (WS).

The first step: set the operating characteristics of the serial port. In lieu of setup from the main menu, the functions SIOINIT and STIMEOUT will do the job. Then, executing PASSTHRU will allow you to type characters to the serial port. To return to the WS environment, use the <STOP> key. If you need the ASCII character set for all or part of the terminal session, type "SETAPL OFF"; return to APL by typing "setapl on".

```
     ∇SIOINIT[⎕]∇
[ 0]   SIOINIT BR ;⎕IO
[ 1]  ⍝SETS BAUD RATE ON SERIAL PORT
[ 2]  ⎕IO←0
[ 3]  →(~BR∊300,1200)/ERR
[ 4]  0 0⍴(61424,BR,96 0) ⎕SYS 45203
[ 5]  →0
[ 6]  ERR:'???'
```

```
     ∇STIMEOUT[⎕]∇
[ 0]   STIMEOUT SEC ;⎕IO
[ 1]  ⍝SETS TIMEOUT ON SERIAL PORT
[ 2]  ⎕IO←0
[ 3]  ⎕AV[,4 2⍴0 10×SEC] ⎕POKE 878+⍳8
     ∇PASSTHRU[⎕]∇
[ 0]   PASSTHRU
[ 1]  ⍝JUMPS INTO ¨PASS THROUGH¨ MODE
[ 2]  0 0⍴⎕SYS 45173
```

```
     ∇SETAPL[⎕]∇
[ 0]   SETAPL B
[ 1]  ⍝SELECTS APL CHARACTERSET ¨ON¨ OR ¨OFF¨
[ 2]  →(~B∊0,1)/ERR
[ 3]  0 0⍴(B+1) ⎕SYS 45194
[ 4]  →0
[ 5]  ERR:'???'
```

```
     ∇ON[⎕]∇
[ 0]   R ← ON
[ 1]  R←1
     ∇OFF[⎕]∇
[ 0]   R ← OFF
[ 1]  R←0
```

(*TO SWITCH TO ASCII, TRY: SETAPL OFF*)

The prime reason to be in the APL WS, communicating with a host computer, is not to use the passthrough mode; rather, it is to drive the serial port from an APL program. This opens up a whole new range of possibilities in communicating with host computers that are simply not possible as a terminal. To this end, I cannot emphasize enough the importance of obtaining an intelligent modem. The examples below are based on a Hayes Smartmodem. For example, to dial (by touch-tone), the number 999-1234, you send the ASCII string "AT DT9991234" to the serial port. If you add a semicolon to the end of the string, the modem returns to command mode, after it has made a connection. With this feature, you may develop an APL-based "Rolodex". Both modem and telephone can be connected (with a Y connector) to the phone jack at the same time: use APL and the modem to place the call; then pick up the phone and talk!

There are other applications as well. I now pay all recurring bills with this type of modem. Several banks and S&L's here let you pay bills via touch-tone dialing. With the payee codes stored in a WS, I use the modem to dial the compu-

ter, and then dial the sign-on codes, payee codes and amounts until I've paid my bills. The key is to control the modem with an APL program. Note that the modem expects ASCII (more exactly, the sequence of binary codes that correspond to the ASCII characters). In APL mode, you need to check on the correspondence between APL and ASCII characters before you decide which APL character to send; refer to the character code tables in Appendix C, the APL manual. In APL, most keys send the required ASCII binary codes if used "as if" they were still ASCII keys. But not all! For example, while SHIFT 3 will send '#' in ASCII, the asterisk '*' has moved to the SHIFT 8 key! To assure correct translation, first find the ASCII character you wish to send in Appendix Table C-3 (p. 107). Note its hexadecimal code; then turn to Appendix Table C-1. Use the APL character with the same hex code.

The set of APL functions below will dial up a host computer from the APL WS. I assume that the host needs only two items: (1) a log-on message and (2) a password. The function LOGON sets up the serial port, using the functions listed above, and opens the serial port. The first message sent to the serial port is modem-specific; it tells the Hayes modem not to echo characters back to SPET. (This isn't really needed unless in passthrough mode). Executed next are three APL functions that dial the host computer and provide the log-on and the password. I prefer to keep these as separate APL functions in the WS, but this is largely a matter of taste; it is always useful when an "emergency" arises.

```
      ∇LOGON[□]∇
[  0]    LOGON ;I;ANS;CMDS
[  1]   ⍝ROUTINE TO DIAL UP IPSA
[  2]   SIOINIT 1200
[  3]   STIMEOUT 15+I←0
[  4]   OPENSERIAL
[  5]   NOECHO
[  6]   CMDS←3 8⍴'DIALHOSTSIGNON  PASSWORD'
[  7]   S1:'CMD: ',REVERSE CMDS[I←I+1;]
[  8]   '  (ENTER "Y" TO SEND)'
[  9]   ANS←⍞
[ 10]   →('Y'≠1↑ANS)/0
[ 11]   ⍎CMDS[I;]
[ 12]   →(3>I)/S1
```

```
      ∇OPENSERIAL[□]∇
[  0]    OPENSERIAL
[  1]   'SERIAL' □CREATE 10
      ∇NOECHO[□]∇
[  0]    NOECHO
[  1]   'α~ ∈0' □PUT 10
      ∇DIALHOST[□]∇
[  0]    DIALHOST
[  1]   'α~ ⌊~9991234' □PUT 10
      ∇SIGNON[□]∇
[  0]    SIGNON
[  1]   'LOGON ME' □PUT 10
      ∇PASSWORD[□]∇
[  0]    PASSWORD
[  1]   'CRYPTIC' □PUT 10
```

These functions do not use the serial port interactively. As best I can determine, you cannot write an APL function that is fast enough to use the port this way. Ideally, I could send a message to the host and then "listen" to the serial port and accumulate characters as they came back from the host, until a carriage return or a long pause signified that my message was complete. The serial port has no buffer, however, and characters are lost unless they are captured as they arrive. While you may write an APL program that continually "gets" characters from the serial port, checks to see if they are indeed characters, and adds them to a string if they are, the program will be too slow; it will lose characters.

Some other APL implementations provide a system function to do the job (APL*PLUS /PC, for example); it allows a message to be sent to the host, under a variety of translation schemes; it then accumulates characters (to a maximum). The function can be initialized so that control returns automatically to the APL program on receipt of any specific characters (such as BELL). This capability is

also needed to interact with other devices, such as Hewlett Packard plotters. The HP 7470A, for example, can be queried for information about the location of the pen, etc., but the response is now very hard to trap from APL. Something along these lines is needed on SuperPET, and it's up to some dedicated 6809 programmer to provide it! Help!

In the example above, I have allowed the user to control when the next message is sent to the host. Listen to the modem and watch the status lights; it usually is clear when the next transmission should begin. This assumes, however, that everything is working smoothly. If the system or network is down, the messages you receive will be quite different from the ones you expect. Since you cannot tell what the message says, only that there is one, this can lead to problems. Hence, after you think you have successfully dialed up the host, use the pass-through mode to check on a successful log-on before you proceed with your task.

One such task might be to upload an APL function to the host. It is actually easier to edit a function locally than on a mainframe, and it costs a lot less. The example below provides a way to upload APL functions from the micro to the host by first opening the function editor on the host and then sending successive lines of the function. This represents the crudest form of uploading. If I could, I would listen after each line is sent until the host sends a prompt for the next line. Barring that capability, I simply delay two seconds between each transmission. This is usually time enough for the host's editor to respond.

```
      ∇SEND_FN[□]∇
[  0]   SEND_FN ;FN;MAT;N;I
[  1]  ⍝UPLOADS APL FN TO HOST APL FN EDITOR
[  2]  'ENTER: ',REVERSE 'FUNCTION NAME'
[  3]  FN←⍞
[  4]  →(3≠□NC FN)/ERR
[  5]  N←1↑⍴MAT←□CR FN
[  6]  I←0
[  7]  SEND '∇',FN
[  8]  S1:0 0⍴□DL 1
[  9]  SEND MAT[I←I+1;]
[ 10]  →(N>I)/S1
[ 11]  □DL 1
[ 12]  SEND '∇'
[ 13]  'SENT'
[ 14]  →0
[ 15]  ERR:'FUNCTION: ',FN,' NOT FOUND'
```

```
| THIS REPRESENTS A VERY SLOW
| WAY TO UPLOAD APL FNS
| TO THE HOST. AFTER SENDING
| EACH FN, YOU NEED TO USE
| PASSTHRU AND CHECK ON THE
| FN IN THE HOSTS' WS. THIS
| IS OBVIOUSLY QUITE TEDIOUS
| AND SHOULD ONLY BE USED
| OCCASIONALLY.
---------------------------
```

```
      ∇SEND[□]∇
[  0]   SEND MSG
[  1]  (□XR MSG) □PUT 10
```

There is no guarantee that all characters sent out the serial port will arrive safely at the host. Is there a better way? The answer is yes. Waterloo has provided powerful communication facilities (HOSTCM) in each of the languages. This includes error checking and automatic retransmission. To use these facilities, however, the proper software must be running on the host computer. Fortunately for APL users, John Wilson has mimicked much of this software in APL. Thus, if you can communicate with a mainframe running APL, you can utilize some of SPET's built-in HOSTCM facilities. This will be the subject of a later article. In the meantime, if you do not have a HOSTCM Specifications Document, order one from Waterloo. The one I have is written by T. Wilkinson and is dated February 1982.

The last example, listed below, provides a way to read files you have downloaded to disk using a terminal program. Since nonprintable characters may have been

transmitted and logged to disk, each record is "cleansed" after being converted to internal APL format. This step runs slowly in APL, since booleans are actual-

```
      ∇READ_LOGFILE[⎕]∇
[ 0]    READ_LOGFILE ;⎕IOERRSTOP;FN;NL;N
[ 1]   ⍝READS FILE "LOGGED" BY WTE
[ 2]   ⎕IOERRSTOP←0
[ 3]   ⍞←'ENTER: ',REVERSE 'FILE NAME'
[ 4]   FN←⍞
[ 5]   ('(T)',FN) ⎕TIE 1
[ 6]   →(0≠⍴⎕STATUS)/ERR
[ 7]   S1:NL←0
[ 8]   CLEAR
[ 9]   REVERSE CENTER 'FILE: ',FN
[ 10]  R1:⎕←CLEANSE ⎕IR ⎕GET 1,100
[ 11]  →(20>NL←NL+1)/R1
[ 12]  N←⎕AV⍳PAUSE ⎕TC[3],'USE PF 3 OR . TO MOVE TEXT, ELSE QUIT',⎕TC[7 4 4 8]
[ 13]  →(132 140=2⍴N)/R1,S1
[ 14]  →EXIT
[ 15]  ERR:'FILE NOT FOUND: ',FN
[ 16]  EXIT:⎕UNTIE 1

      ∇CLEANSE[⎕]∇
[ 0]    R ← CLEANSE CHARS
[ 1]   R←(CHARS∊⎕AV[⎕IO+13+⍳113])/CHARS
```

```
      ∇REVERSE[⎕]∇
[ 0]    CR ← REVERSE C
[ 1]   CR←⎕AV[128+⎕AV⍳C]
      ∇CLEAR[⎕]∇
[ 0]    R ← CLEAR
[ 1]   R←⎕TC[⎕IO+4]
      ∇CENTER[⎕]∇
[ 0]    R ← CENTER MSG
[ 1]   R←79↑((⌊(79-⍴MSG)÷2)⍴' '),

      ∇PAUSE[⎕]∇
[ 0]    X ← PAUSE MSG
[ 1]   '(F:1)KEYBOARD' ⎕TIE 2
[ 2]   ⍞←MSG
[ 3]   R:X←⎕GET 2,1
[ 4]   →(⎕AV[⎕IO]=X)/R
[ 5]   ⎕UNTIE 2
```

ly implemented as floating point variables; you may choose to bypass the process if you are confident that the file is in good shape. Cleansing removes all line-feeds and backspaces; overstruck characters from the mainframe that are not supported by SPET's character generator show up as two characters.

6425 31*ST ST., N.W., WASHINGTON, D.C.* 20015 *U.S.A.*

ATTENTION 8050 OWNERS! Fixes for the DOSbug There's a serious bug in 8050 drives, which we've encountered only on Tandon-made 8050s. These drives are identified by a top-closing door hinged at the top. If the problem exists for Micropolis-made 8050s or 8250s, it has not been reported. John Frost of Seattle defines the problem and one solution. We've found several more ways to cure the problem (which is infrequent. We've had it only five times in two years). John writes:

"A bug in the DOS sometimes prevents reading or writing to disk. The drive motor starts and the drive attempts to read a file, but then fails to do so. It finally times out with the error message: "DRIVE NOT READY." The problem apparently occurs when power is removed from a drive after a disk access to a track larger than 55. The DOS can't recover control of the R/W head on the next power-up.

"You can recover from this condition if you open and quickly close a drive door as the drive attempts to read a directory. The drive responds to this error by 'homing' the R/W head. You get lots of error lights, the drive makes some awful sounds, and generates an error message. If you clear the error message, the drive is ready for operation. For info on this and other bugs/idiosyncracies of our machine, I recommend the book <u>CBM Professional Computer Guide,</u> by Osborne and Jim and Ellen Strasma (Osborne/McGraw-Hill)."

Later, John reported that the door-flip did NOT work when he again had the bug. Steve Zeller reports that if you always call for a directory on both 8050 drives before you shut down, you'll not face the problem on subsequent start-ups.

Ye ed has encountered the problem several times on a Tandon-made 8050, always at menu, at start-up. John's door-flip approach didn't cure it, nor did the method recommended by AB Computers from BASIC 4.0 (left). So we loaded the mED from our 4040 drive, and found that both INITIALIZE and VALIDATE, given with a prefix of: g ieee8-15., will kick the 8050 to proper operation. Later, we found that MOUNT also worked. But: how do you load the mED if you have no extra drive? The answer, as with the invention of the wheel, was so obvious we needed three weeks to think of it: drop into the monitor and give the MOUNT code!

```
open 1,8,15
print#1,"I0"
print#1,"I1"
close 1
```

```
>m 1000 cc 10 07 bd b0 e7 3f 64 69 73 6b 2f 30 00     <RETURN>
      (Code for drive 0. After you enter line above, 'go' on the next line.)
>g 1000    <RETURN>
```

Change the last two bytes to read: 31 00 to MOUNT drive 1, and 'g 1000' again. We know this works, because we've twice used the code in the monitor to recover.

◇◇

UDUMP - A UNIVERSAL DUMP TO IEEE4, PRINTER, SERIAL, or DISK

For six months, we've had a number of great dumps in hand from Gary Ratliff and Terry Peterson--but all of them dumped to just one file, either to disk, printer, serial or ieee4. When in the monitor with a dump to printer, we needed a dump to disk, etc.--the right one never was in memory. Terry P. has been working hard on a universal dump which'll load up in Bank 15 --but the operating system neglects to handle interrupt-driven routines in the banks when you load a new language. While Terry was working on that one, we converted a routine Terry and Gary had written/revised into a universal dump. With UDUMP loaded, you always can dump to disk, serial, ieee4, or printer.

It is reliable; better, it dumps from screen line 1 through the line the cursor is on and then stops. Put the cursor on line 25, and it dumps the whole screen.

```
     xref curpos_, memend_, conbint_, printf_, openf_, kyputb_
     xref closef_, fputchar_, fputnl_, initstd_, intvctr_, kyindx_, kyptr1_
     xref kyptr2_, file_, usirq_, printer_, serial_, write_, append_

main    equ   *
        jsr   initstd_
        ldd   #main
        std   memend_
        clr   $32
        clr   busy
        ldd   #instr
        jsr   printf_
        ldx   #8
        ldd   intvctr_,x
        std   resvd
        pshs  x
        ldd   #start
        jsr   conbint_
        puls  d
        rts
```

Load this program from main menu; any time thereafter you can dump your screen to any printer or to disk by pressing PF7 (SHIFT KEYPAD 7) and by touching the appropriate key: 'i' for ieee4, 'd' for disk, 'p' for Commodore printer, or 's' for serial. No RETURN is required after you touch the key; the character is 'got' from the keyboard.

You will not scroll any lines off the top of the screen if you dump from line 25.

Put a disk file named 'file%n' on drive 0, since all dumps to disk are appended to that file.

Delton P. Richardson kindly revised the code so that reverse-field characters on screen are sent

```
start   equ   *
        jsr   [resvd]
        ldb   kyindx_
        cmpb  #$95
        lbne  done
        tst   busy
        if eq
          dec   busy
          ldd   #margin
          jsr   printf_
          ldd   curpos_
          addd  #80
          std   where
          ldd   kyptr2_
          std   kyptr1_
          bsr   getit
          ldb   #'d
          guess
            cmpb  type
            quif  ne
            ldd   #append_
            pshs  d
            ldd   #file_
          admit
            ldd   #write_
            pshs  d
            ldb   #'i
            cmpb  type
            quif  ne
            ldd   #ieee
          admit
            ldb   #'s
            cmpb  type
            quif  ne
            ldd   #serial_
          admit
            ldd   #printer_
          endguess
          jsr   openf_
          std   outpt
          puls  d
          if ne
            ldx #1
            loop
              ldb  kyindx_
              cmpb #4
              quif eq
              pshs x
              bsr  print
              puls x
              leax 80,x
              cmpx where
            until hs
            ldd outpt
```

to dot-matrix printers as normal characters, and also suggested several changes in an early version which make UDUMP much more versatile.

We did not have time to revise the code to reject any characters but the four listed above, so the dump defaults to 'printer' if you press the wrong key.

This is a reliable dump, very useful in the monitor, since you may send readouts both to disk and to printer.

Much of the original code was written by Jeff Larson, Gary Ratliff, and Terry Peterson. Without that base, we'd have been lost. A few interesting things did pop up:

System routine kyputb_ (See Ratliff's notes, p. 83, No. 7) 'gets' characters from the keyboard without a RETURN, and sticks them in the keyboard buffer ($130 to $157) without putting them to the screen (it returns the character in B register).

That routine can cause a serious problem in the monitor. On the next-to-final version, we dumped monitor data well, but every so often the keyboard buffer dumped its contents onto the screen with all the CR's it held. We'd get a series of INVALID COMMAND notes from the monitor as it rejected the buffer input, or would crash when the command was valid (>g 7e70 will crash you if you have already 'gone'!). We tried a number of solutions, but we found that the only reliable way to whip this problem was to clear the keyboard buffer after each run. See comments on the code just above the label 'done'.

For those who need a linefeed (LF) with every CR, we've added some optional code at the end of the subroutine 'print'. If you use it, it sends an LF to all printer options, but sends single-spaced material to disk.

Add the following 'usrlib.exp' file to your language disk before linking, since we use a number of routines which are not in watlib.exp.

File usrlib.exp:

```
export memend_   = $22
export intvctr_  = $0100
export curpos_   = $0122
export kyindx_   = $012b
```

```
          jsr closef_
        endif
        clr  busy
      endif
      ldy   #$130
      loop
        clr   ,y+
        cmpy  #$158
      until eq
done  rts

margin fcb 11,13,0

where rmb 2

getit loop
        jsr   kyputb_
        cmpb  #0
      until ne
      stb   type
      cmpb  #$7b
      if pl
        jmp getit
      endif
      rts

print lda   #79
      leax  $7fff,x
      loop
        ldb a,x
        cmpb #'
        quif ne
        deca
      until mi
      cmpb  #'
      if ne
        loop
          ldb ,x+
          cmpb #$a0
          blo  norm
          subb #$80
norm      pshs d,x
          ldd outpt
          jsr fputchar_
          puls d,x
          deca
        until mi
      endif

      ;ldb   #'d
      ;cmpb  type
      ;if ne
        ;ldd outpt
        ;jsr fputnl_
```

```
      export kyptr1_   = $012c
      export kyptr2_   = $012e
      export usirq_    = $02ff  (File continued below)
;-------------------------
; Start of keyboard buffer.
; Characters w/out CR 'leak' to screen...and enter
; false monitor commands, so clear buffer to the
; end, at $157, on each pass.
;-------------------------
      export printer_  = $b129  (continued file)
      export serial_   = $b144
      export write_    = $b191
      export append_   = $b19e
      export kyputb_   = $dd82
      export file_     = $f451   (end, usrlib.exp)
```

The .cmd file for UDUMP without an extra linefeed:

```
"udump"
org $7e70
include "disk/1.usrlib.exp"
include "disk/1.watlib.exp"
"udump.b09"
```

Exactly two bytes are free below $8000 after UDUMP is loaded into $7e70. If you add to the code, best move it down. We also note that the chunk of code incorporating the GUESS...ENDGUESS is near block maximum, so any attempts to add much to it will fail unless you go to a subroutine.

If you add the extra linefeed for printers which need it (below), set origin to $7e60!

Don't try to load this one in the monitor--it RTS's to menu. If you want to, take out the CLR $32 in MAIN and substitute SWI for RTS in MAIN. You should also remove INSTR (the prompt) and the two lines in MAIN which load and print the prompt. Yes, it loads in the monitor that way and runs--but the stack is utterly mad, the registers are equally insane, and we firmly expect SuperPET to implode. Proceed at your own risk.

```
;-----------------------------------------
; OPTIONAL printer code for linefeed with each CR:
; Remove semicolons and use the next 6 lines.
; Is this a dump to disk?
                                  If you use this, set
; No, not a disk dump.            origin to $7e60!
;
; So, add a linefeed for printer.
```

```
        ;endif              ;---------------------------------------------
        ldd outpt           ; Always use this line.
        jmp fputnl_         ; Ditto

outpt   rmb 2
resvd   rmb 2
busy    rmb 2
type    rmb 1

ieee    fcc "ieee4"
        fcb 0               ; No, we aren't careless--the double %n%n in the
                            ; prompt below double spaces the screen lines.
instr fcb  12
      fcc "UDUMP loaded. To use: press PF7, then one of keys below.%n%n"
      fcc "To disk: 'd'¦ to ieee4: 'i'¦ to printer: 'p'¦ to serial: 's'%n%n"
      fcb 0
      end
```

◇◇

PROBLEMS IN MPASCAL AND MFORTRAN
Crashes in Recursion and Errors in Large Numeric Values

Thomas K. Roney teaches at Kent School, Kent CT, 06757, and asked why SuperPET crashed in lengthy recursions and why a factor of ten was missing in large floating-point decimal displays. We asked Bob Davis, Associate Editor in mPASCAL, to look into it, and he did. His reply to Mr. Roney is printed below:

"By the way, I've discovered that the floating decimal point display bug exists in mFORTRAN also! Recursing functions can be nested 10 deep and recursing procedures can be nested 16 deep without a problem in microPASCAL. And, yes, SuperPET does go into rigor mortis (catatonia?) beyond these limits without any error message. You may avoid the problem only with a software solution--write the recursing program to tally the calls; if the maximum is reached, provide an error message and prevent further recursion.

"I find that numbers up to but not including 1000000000.0 (0.1E+10) are displayed correctly in floating decimal format. If there are no more than nine digits to the left of the decimal point, the display will be correct. However, any larger number will be displayed a factor of ten less than it should be in floating decimal point format, as you discovered. Again, you must provide a software solution. In mPASCAL, this might be: if value < 0.1E+10 then writeln (value :15:3) else writeln (value * 10.0:15:3).

"MicroPASCAL will accept floating decimal point input correctly up to 1 followed by 37 zeros, a decimal point, and a zero! Beyond this, you receive an error message, "Warning;line nn: Real number overflow" and the program continues although the value of the variable is garbage. Also, the maximum field width for a floating decimal point display is 20. Neither of these limitations should normally be a problem, however.

"Out of curiosity, I tried mFORTRAN using Example 27 from the tutorial disk to see if the decimal shift bug is unique to mPASCAL or is general to the Waterloo system. I find that the same decimal point shift occurs using the corresponding format; i.e., mFORTRAN's 'print f15.3,r', which is equivalent to your example in mPASCAL of 'writeln (r:15:3)'. You should warn your mFORTRAN students of this and have them program a software solution.

"The decimal point shift bug does not exist in microBASIC, since numbers 0.1E+10 or larger (and 0.1E-4 or smaller) are automatically converted to scientific notation. Example 12 on the tutorial disk illustrates this. Someone else will have to answer if the bug exists in APL or COBOL, but I don't believe it can."

Later, Bob told us that the 'factor of 10' problem affects only the screen display. Since internal calculations are right, correct the display <u>only.</u>

PLOTTER PROBLEMS
In SuperPET

H.O. Pritchard, Professor of Chemistry at York University at Downsview, Ontario M3J 1P3, reports problems in mFORTRAN while using an HP7470A plotter with SuperPET. The plotter is addressed as ieee5. When, after mFORTRAN has been running for a long time, he requests digitized input from the plotter with the sequence shown at left, he receives an EOF error. The problem is cured either by reset of SuperPET, or by revising the program to read as shown in the second version at left. Dr. Pritchard states that the problem afflicts a number of SuperPET models, from the earliest to the latest, and suspects ROMs carrying IEEE system routines are in error, since the problem does <u>not</u> occur when he uses BASIC 4.0 with the 6502. He reports that the same problem crops up in microBASIC and in Assembly language. Waterloo has been queried and we'll report on their response. Note: this is an <u>input</u> bug; it may afflict other input devices on the IEEE bus. Dr. Pritchard also also reports the same problem noted by Bob Davis elsewhere in this issue: the loss of a factor of ten in screen displays of large numeric values in mFORTRAN.

```
open (unit=10,file="ieee5")
read (10,*) x,y,ipen
   [Returns EOF error]

open (unit=10,file="disk5"
read (10,*) x,y,ipen
       [Works]
```

MORE ON RELATIVE FILES
by Loch Rose
102 Fresh Pond Parkway
Cambridge, MA 02138

In this article, I'll cover some less basic aspects of relative files, beginning with ways to handle files if some records contain more data items than others. Program 1, below, creates such a file:

<u>Program 1</u>

```
400 randomize : string$ = "abcdefghijklmnopqrstuvwxyz" : c$ = ","
410 terminator$ = chr$(255)        ! terminator$ could be any character(s)
420 open #2, "(f:60)examples,rel", output
430 for recnum = 0 to 20           ! write 21 records
440     num_items = int(rnd*5)     ! each record randomly contains 0-4 items
450     print #2, rec=recnum, "";  ! positions you at start of rec. #recnum
460     for j = 1 to num_items     ! if num_items=0 this loop is skipped
470         print #2, str$(string$,rnd*21+1,4); c$;    ! print a 4-char. string
480     next j
490     print #2, terminator$      ! last char. (since no ';', end of record)
500 next recnum
510 close #2 : stop                ! SAVE this program
```

As usual, data items are separated by commas; note that now the last data item is always a chr$(255). The remark on line 490 refers to the fact that a PRINT# statement that does not end with a semicolon causes the end of the current record and the start of a new one, though in this case line 450 would cause you to move to the next record on the next iteration anyway. Program 2, below, shows how you can INPUT the file created above:

<u>Program 2</u>

```
200 open #2, "(f:60)examples,rel", input
210 for recnum = 3 to 18                 ! for fun, only look at records 3-18
```

```
220    num_of_data = 0                  ! we'll count # data items per record
230    input #2, rec=recnum, data_item$  ! only this 'input#' has a 'rec='
240    while data_item$ <> chr$(255)    ! input until terminator character
250       num_of_data = num_of_data + 1
260       print tab(10*num_of_data-9); data_item$;
270       input #2, data_item$          ! note that this item still comes from
280    endloop                          !    same record (i.e. record #recnum)
290    print tab(51); "# items in record"; recnum;"="; num_of_data
300 next recnum
310 close #2:stop                       ! I suggest SAVEing this program
```

The program keeps inputting data items from a record until it encounters a chr$(255).

Another way is to store the number of data items in a record at the start of the record itself. We'll create such a file by modifying two lines in program 1; the new lines should read as shown below. (Note: line 450 also stores the record # at the start of each record.) Before running the new program (called #3), scratch the old file by typing 'scratch "(f:60)examples,rel"'. I emphasize that you must ALWAYS refer to a relative file by its full name, '(f:60)' and all, or horrible things can happen.

```
450 print #2, rec=recnum, recnum; c$; num_items; c$;
490 print #2
```

Program 4 reads the new data file (in reverse order, for variety). You'll notice that it is actually a bit faster than program 2, despite having to read one more piece of data per record, so I'd give this method the edge:

Program 4

```
600 open #2, "(f:60)examples,rel", input
610 for recnum = 18 to 3 step -1            ! read in reverse order
620    input #2, rec=recnum, filerecnum, num_of_data  ! read first 2 data
630    print filerecnum; tab(11); num_of_data;
640    for i = 1 to num_of_data             ! read # items we know are there
650       input #2, data_item$
660       print tab(10*i+11); data_item$;
670    next i
680    print
690 next recnum
700 close #2:stop
```

I don't bother with error trapping in any of these programs, as SuperPET catches most errors without being asked. (One error it won't catch--try to input more items from a record than it contains and SuperPET gets the extra items from the next record.) There is one special use of IO_STATUS: if you use GET# to read a relative file, IO_STATUS changes from its usual value of 0 to 1 at the end of each record. This is your only way to keep records distinct, as GET# cannot use the 'rec=' clause. I prefer to take the whole record at a gulp with LINPUT#, and avoid GET# entirely.

In the (frankly) unlikely event that you wish to break records up into a known number of fields, each field containing an unknown number of data items, you can use the LINPUT# statement. LINPUT# reads everything up to the end of a record or up to a chr$(13) character, whichever comes first. In the following program, we break each record into three fields:

Program 5

```
 10 c$ = "," : CR$ = chr$(13)         ! CR$ will separate the three fields
 20 open #2, "(f:100)check,rel", inout
 30 for i = 1 to 5                    ! create five records
 40    print#2,i;c$;i*i;CR$;value$(i);c$;"record #";i;CR$;"this is it";CR$
 50 next i                            ! last CR$ is optional
 60 linput #2, rec=3, a$              ! input first field, record #3
 70 linput #2, b$                     ! input second field, same record
 80 linput #2, c$                     ! input third field
 90 print a$ : print b$ : print c$    ! print the three fields
100 stop
```

[Ed. You'll find what Loch writes most useful, but only if you enter the programs, run 'em, and revise 'em. As Gary Ratliff said, early on, "You learn to program by programming, not by reading about it." We'll print a method by Loch, next issue, which very cleverly lets you specify any date after 1900 and learn where the record for that date is kept in a relative file.]

◇◇

HOW MANY RELATIVE FILES DID YOU SAY, STAN?

We got a note from Stanley Brockman, 11715 W. 33rd Place, Wheat Ridge, Colorado, 80033, saying he was happy he innocently didn't know he was supposed to be limited to 720 records in relative files on his 4040 drive, else his primitive data base manager program wouldn't have more than 1000 records in it. Stan went through his disk manual, and calculated he should be able to get 2088 records (as opposed to the 720 on p. 80 of the Gazette). He tested and confirmed that number. Later, he wrote that on the CBM side you can get one more relative record: 2089, though he isn't sure why. We double-checked Stan's figures and stuffed REL records on the drives listed below, until they would not take more. The capacities on 80-byte records in SPET most certainly are what Stan says they are. We read good files through the numbers listed below:

int(658*254/recordlength)-1
= 2088 records

	Micropolis 8050	Tandon 8050	Tandon 4040
Records	2285	2285	2088
Blocks	726	726	664

* * *

ON EDITING REL FILES

You can pull 1068 80-byte records into the microEDITOR from a REL file before memory fills; if you have this number of records or less, it's a nice way to get a list which correlates record number with the data in that record. You can also get records larger than 80 bytes into mED for the same reason, but you must specify the record size if you try, as in the 'get' at left. You can recover a REL file in 'fixed' filetype as a TEXT file, but the format is mucked up. If you refile an edited REL file from the mED as a RELATIVE file, despite the record size you may specify (as in f:200), you'll never get it back in 200-byte chunks--but rather in lines of 80 bytes. In language, if you ask for input of rec = 20, you'll get back the contents of the twentieth line in the file (assuming option base 1).

g (f:200)file,rel
(t: works)

There is one exception: you can edit and refile records of 80 bytes, because in such files one line = 80 bytes. Otherwise, don't refile REL files from the mED; you ruin the relative access to the file. (If you do refile it, specify it as a relative file, as at left.)

p file,rel

SUPERPET USERS

%%%°%%%&%%%.%%%$%%%°%%%&%%%.%%%$%%%°%%%&%%%.%%%$%%%°%%%.%%%&%%%$%%%°%%%.%%%&

Have you ever wondered what the /%*%$%& is the difference between 'c*/ %*//' and '*c/ %*//' ? Tired of flipping that switch just to do a 'collect'? The SuperPET Tutorial Disk reveals the mysteries of the data editing commands and 'meta-character' strings, using clear and useful examples. It also contains:

A GENERAL NARRATIVE DESCRIPTION OF THE MICROEDITOR

- SYNTAX AND EXAMPLES FOR ALL MICROEDITOR SEARCH STRINGS.
  EXAMPLES AND EXPLANATIONS OF ALL MICROEDITOR COMMANDS.
  EXAMPLES AND EXPLANATIONS OF ALL MICROMONITOR COMMANDS.
  EXPLANATIONS OF ALL SETUP MENU OPTIONS.
  COMPLETE INFORMATION ON THE PROGRAMMED FUNCTION KEYS.

- INFORMATION ON ALL FILE TYPES AND FORMATS

- EXAMPLES OF ALL VARIATIONS OF THE DISK ACCESS COMMANDS.

- INSTRUCTIONS ON ISSUING ALL DOS COMMANDS FROM THE EDITOR.
  EXPLANATIONS OF ALL DOS ERROR MESSAGES.
  INSTRUCTIONS ON AUTOMATING DISK MAINTENANCE TASKS.

- INFORMATION ON RS-232C AND THE TERMINAL FACILITIES.

- A TABLE OF IMPORTANT SYSTEM ADDRESSES AND SOFTWARE SWITCHES.

- DECIMAL AND HEX VECTOR ADDRESSES OF WATLIB AND FPPLIB ROUTINES.
  6809 ASSEMBLER INSTRUCTION OPCODES, MODES, AND LENGTHS.
  HEXADECIMAL-DECIMAL CONVERSION TABLE.

- HEXADECIMAL AND DECIMAL ASCII CHARACTER TRANSLATION TABLES.

THIS PRODUCT COSTS ONLY $39.95, POSTAGE AND HANDLING INCLUDED. THE ITEMS MARKED WITH '∘' ARE ALSO AVAILABLE ON A REFERENCE CARD WHICH IS INCLUDED WITH EACH TUTORIAL DISK ORDERED. THE REFERENCE CARD ALONE COSTS ONLY $10.

IF YOU ORDER ANY DISK-BASED PRODUCT, THE DISK YOU GET WILL ALSO CONTAIN A SELECTION OF THE BEST PUBLIC-DOMAIN SUPERPET SOFTWARE FROM VARIOUS SOURCES. ALSO AVAILABLE IS THE APL-MICROEDITOR INTERFACE; IT ALLOWS USE OF THE MICROEDITOR FOR EDITING APL FUNCTIONS AND VARIABLES. VOLUME DISCOUNTS ARE AVAILABLE (30 PERCENT OFF FOR 2-10; 40 PERCENT OFF FOR 11-100.

SEND A CHECK NOW (AND SPECIFY 4040 OR 8050 FORMAT); OR WRITE FOR INFORMATION TO→

DYADIC RESOURCES CORPORATION
PO BOX 1524, STATION A
VANCOUVER, B.C. CANADA V6C 2P7

('c*/ %//' hangs up; '*c/ %*//' does nothing; '*c*/ %*//' removes spaces from left)

to set up printer files for any printers not included. We merged 'CLIP with the disk printer file for DIABLO and were printing in five minutes. Once the printer file is merged, you needn't ever set up for that printer again, for you file the merged version to disk. We have two such files: clip.diablo, and clip.ascii, the last being most handy for output of printer files to disk for telecom and mED in 6809. If you have the wrong version loaded, you can stuff another printer file into 'CLIP just before you print--quickly.

Ranges in 'CLIP can be set to lines, words or characters, not just to lines, as in WordPro. A range is reverse field set by you to mark text you want to erase, delete, move, or save...A lot of complex commands/options come to screen with a YES/NO prompt; the most common response is pre-printed so you can hit a RETURN and accept it...You can change device numbers on disks/printers within 'CLIP...Batteries provides character generators for foreign languages which you can employ from 'CLIP.

You can search/replace five phrases at once (want to??); but after you find a phrase, you have the option to replace/not replace (bingo!)...'CLIP provides a multiple line insert--tell it how many blank lines you want; it creates them, complete with hard carriage returns... **AND** the best trick yet: you can jump instantly to end-of-text with SHIFT/RUN, and need not spend all day Saturday cursoring down...'CLIP deletes a range of text quickly, far faster than WordPro... You can COPY any phrase on the screen to the screen up to 255 times (superb for creating forms)...and 'CLIP makes perfect ASCII files with the 'True Ascii' file on the 'CLIP disk--without the prefixed quotation marks WordPro creates...If you want semi-proportional spacing, 'CLIP supports it, but you must create your own printer file, using the 'CLIP instruction manual.

And more: In formatted output, you get a word-count of each page...And there is a built-in sort for lists... Plus a simple way to create a Table of Contents, which is automatically saved to disk. You mark all entries you want for the table; 'CLIP saves that entry and its page number...And you can restart output at the top of any page if you catch an error...You can move, erase, or reproduce a column in text...And you can add/subtract both rows and columns of digits, to a specified number of decimal places. Plus some other goodies. As might be expected, you must learn a gob of new commands, but you cannot get power without complexity. Those who need only a text-processor for simple jobs will find 'CLIP as easy to use as WordPro in that context; switchover to 'CLIP commands is easy.

An integral spelling checker may be available about March '84, with a dictionary of 20,000 words, and space for another 5,000 on a 4040 disk. Reported time to check a full screen page (over 700 lines): about 150 seconds.

Want to update to newer versions of 'CLIP? Keith Hope, the Technical Director of Batteries, writes: "Current owners can update their program to the newer version several ways. They can request a copy of the new diskette from their local Commodore dealer, or they can send $15.00 to Batteries Included and we will send them an updated diskette." That is a responsive and sensible arrangement.

We've covered the good changes--those that particularly impressed us. There are some things we don't like:

1. The directory readout in 'CLIP is a disaster; it mars the whole program. If there is a way to send one to printer, we couldn't find it. 'CLIP returns to

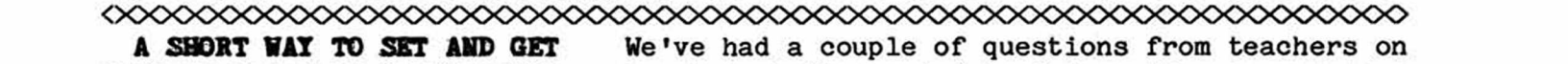
the old Commodore single-column string-of-spaghetti, a hopeless arrangement for professional word-processing. While you can pause the directory and call for a file, you can't restart and rescroll. You get to call for a directory--from the top--all over again. WordPro's four-column directories are far superior, for you can edit them, scroll them, send them to printer, xerox them, and file them on 8.5x11 paper. If you could print 'CLIP directories, you'd have a yard of sphaghetti on 8050s and 8250s. (Batteries says Version 9000B will let you load directories as text, which means you can sort, print, and edit--in four columns, we hope!). Meanwhile, we print 'CLIP directories from WordPro.

2. In 9000A, WordPro's old, quick 3-keystroke 'delete-word, delete-sentence commands are replaced by a slow, involved process in which you set range on the phrase and then 'kill' it. It is so slow we gave up and deleted manually. 9000B is supposed to incorporate a quick 3-keystroke method. It's needed!

3. A serious design deficiency: An HCR (hard carriage return, which shows on screen as a left-pointing arrow) at the start of a line is not deleted when you enter text to its right. (All WP programs we've ever seen do it automatically). Result: when you file to disk and retrieve that line, it's indented half-a-page. Since 'CLIP prints HCR's automatically on new, blank lines, you find it easy to send text to file with an HCR on a line, with disastrous results. This bug must be fixed; we're not about to check every indented line for a prefixed HCR.

4. We loathe blinking cursors, and 'CLIP's cursor is a fast blinker, which drove us out of the house screaming in half an hour. Some folks like blinkers; some don't; there ought to be a way to turn that infuriating wink off.

In sum, we can recommend 'CLIP, with the reservations above, since it sells for about four-tenths of WordPro's price and is a marvellously versatile WP program. Barry Bogart strongly recommends it; Jim Strasma's last words in MIDNIGHT are: "This is it! This is the one you've waited for...." Barry Bogart adds, "I think any company interested in supplying the SuperPET market deserves our patronage."



A SHORT WAY TO SET AND GET TIME AND DATE IN MICROPASCAL

We've had a couple of questions from teachers on how to get time and date information in program in mPASCAL, which has no instrinsic functions to set or get time or date. It's easily done, as demonstrated below by a couple of short programs written by Marvin Cox of 4900 W. 96th St., Oak Lawn, Ill. 60453, which peek/poke the direct memory locations where time/date are kept in SPET. If you want more information on settime/gettime, see p. 46 ff, Vol. 1. Those who need them can easily write independent programs to set or get time or date from the examples and data below.

```
program datesetsee_pd(input,output);  {enters date and then reads it}
var
    ii:integer;
    date:char;
begin
writeln(chr(12));
writeln('Enter 3 characters for month, 2 digits for day, 4 digits for year.');
writeln('Use a space between month entry, day entry, and year entry.');
for ii:=0 to 10 do
  begin
  read(date);              {Date is kept in $0164 thru $016e, in 11 consecutive}
```

```
  poke(356+ii,ord(date))    {bytes (or 356 to 366 decimal, inclusive). Eleven}
  end;                      {characters are maximum for the date set.}
writeln;
for ii:=0 to 10 do
  write(chr(peek(356+ii)));
  writeln
end.
```

* * *

```
program timesetsee_pd(input,output); {Enters time and then reads it.}
var
    timeset,ii:integer;
begin
writeln(chr(12));
writeln('Enter hh,mm,ss,jj without commas, and with a RETURN after each.');
for ii:=0 to 3 do
  begin                     {Time is kept in four consecutive bytes, as follows:}
  read (timeset);           {$160, hours; $161, mins; $162, sec; $163, jiffies.}
  poke(352+ii,timeset)      {This converts to 352 through 355, decimal.}
  end;
writeln('New time is', peek(352),':',peek(353),':',peek(354),':',peek(355))
end.
```

BITS BYTES & BUGS : by Gary L. Ratliff,Sr.

215 Pemberton Drive, Pearl, Mississippi 39208

Let's answer the questions of the last issue. The first: What alterations to code are needed if 'pshs d,x' is used in place of: 'pshs x - tfr d,x - pshs d'. This method saves considerable time, as each push onto the stack takes five cycles, plus one additional cycle for each byte pushed. Pshs x and pshs d, therefore, account for 14 cycles; five for each push, plus two additional cycles for each of the two-byte registers. In contrast, 'pshs d,x' needs only nine cycles.

As to what revisions are needed if 'pshs d,x' is used: As you'll see in the answer to the question of the order of push and pull to the stack, the order of the registers (relative to the stack pointer) remains the same with the new method, so that the 'subd ,s' code needs no change. If the order had been reversed, and the D register had been placed on the stack prior to the X register, then the 'subd ,s' instruction would need revision to 'subd 2,s' to account for this. The original and revised codes are compared at left. The X register and the D register are saved in one step, which kills the need for the 'pshs d' instruction. The answer in D must now be placed on the stack, as it will be recovered by the subsequent pull. This saves some time and also eliminates one line of code. But isn't it more confusing and a little more error prone than the method used originally?

from	to
pshs x	pshs x,d
pshs d	(eliminate)
ldx 2,s	std ,s
leas 4,s	puls x,d

The second question (what is the order of push and pull to the stack?) is best answered by a demonstration. We can't use the system stack for this, since it is used by all interrupts and JSR and RETURN instructions. Instead, we substitute the user stack and show the order in which registers are pushed and pulled. To see this, enter the monitor (from the mED with: mon <RETURN>, or from DEVELOPMENT or main menu with: m <RETURN>). We 'm' for 'modify memory,' starting the code at $1000. The line of code tells the computer to set the U (for User) Stack to

```
>m 1000 ce 20 00 36 ff 3f
```

```
>g 1000
>r
  PC   D    X    Y    U    S   CC DP
:1005 0102 0304 0506 1ff4 0220 c0 00
>d 1ff4-2000
1ff4  c0   Condition Code (CC)    12
1ff5  01   A (D high byte)        11
1ff6  02   B (D low byte)         10
1ff7  00   Direct Page (DP)        9
1ff8  03   High byte, X            8
1ff9  04   Low byte, X             7
1ffa  05   High byte, Y            6
1ffb  06   Low byte, Y             5
1ffc  02   High Byte, S pointer    4
1ffd  20   Low Byte, S pointer     3
1ffe  10   High Byte, PC          2
1fff  05   Low Byte, PC           1
2000  xx                  (Order)
```

start at $2000, to push all registers on the User Stack, and then to break back to the monitor. Before you try this, be sure to overtype the D, X, and Y registers to show the values at left. We need the values to track what happens (don't reset PC or the User Stack Pointer, which will be at 0000 before the 'run'.) Then give your >g 1000. After the run, dump registers as we show at left. Note that the User Stack Pointer (U) has decremented to $1ff4. The stack 'stacks' downward. I have annotated the memory contents from $2000, where we started the stack, to $1ff4, at bottom. The 'push order' of the stack is clearly shown, left. Note: the stack pointer not used (here, the Hardware Stack Pointer), is saved. If we had pushed all this onto the Hardware stack, we would have saved the value in the User Stack Pointer instead. Of necessity, the 'pull order' of the stack is the opposite of the push order.

Thus we see that 'pshs x,d' places the contents of the X register onto the stack before the contents of the D register. The stack pointer shows the last location on the stack which was filled. Hence the pointer is decremented before a value is stored. (And, therefore, subd ,s in the length example subtracts the value of the start address which was originally saved in the D register from the count which was obtained in the adjusted value of the X register.)

The use and misuse of the stack is such an important topic that it will be treated in an article as soon as I can get to it (no more than a year!).

Music generation on the SuperPET is a very complicated topic. The examples presented will generate no sound unless you find and correct the error. In all examples the nop instruction is incorrect. Assembly language operates sooo fast that the ear can't detect the sound presented in one millionth of a second. But the 6809 has a special instruction to handle such events which the 6502 lacks, and you'll find it in the Waterloo Assembly-Language manual (as well as at the end of this column if you get tired of learning while searching). When generating sound with 6502 code, you need elaborate delay routines to obtain the timing which will appear to the ear to be music.

There are specific boards designed for use on the 8032 (SuperPET) which allow the creation of pleasing musical compositions. The capabilities of the Commodore 64 with its SID chip are even more spectacular (my brother owns a 64, and both the color and sound always amaze me. I even purchased a TI-99/4A recently to get first-hand experience with color and sound capabilities). And though I thought that a SuperPET could not have additional boards added, a letter from Col. Stallings states that he has the MTU music board and software.

We are now going to produce a few sounds from SuperPET. I can't even read music, so don't expect a treatise on composing sonatas. The basic information is contained in the book, Programming the PET/CBM, by Raeto Collins West, published by COMPUTE! Books. More information is found in CURSOR (no longer published) and by Gregory Yob in one of his Creative Computing columns.

Our first task will be to attempt to reproduce the familiar chimes which we hear when the SuperPET is turned on in 6502 mode. To do this, we'll translatete the code which runs the chimes from 6502 to the equivalent 6809 code.

```
;chimer.asm--for 6809                    6502 Code
driver jsr chimes
       swi
chimes jsr chime                         e6a4 20 a7 e6  jsr chime
chime  ldy #$10                          e6a7 a4 e7     ldy $e7 ;chime timer loc
       sty $e7                           e6a9 f0 25     beq $e690;contains $10
       lda #$10                          e6ab a9 10     lda #$10
       sta $e84b  ; set VIA timer to     e6ad 8d 4b e8  sta $e84b
       lda #$0f   ; free running mode    e6b0 a9 0f     lda #$0f
       sta $e84a                         e6b2 8d 4a e8  sta $e84a
       ldx #7     ; num. of note 2 play  e6b5 a2 07     ldx #$07
outer  lda  tabl,x                       e6b7 bd 4d e7  lda $e74d,x
       sta $e848 ; play selected note    e6ba 8d 48 e8  sta $e848
       lda $e7                           e6bd a5 e7     lda $e7
inner  leay -1,y ; same as dey           e6bf 88        dey
       bne inner                         e6c0 d0 fd     bne $e6bf
       ldy $e7 ; reset y value           ; when y dec in 6809 it goes to $FFFF--
                                         ; a rather long delay, so we reset here
       sec                               e6c2 38        sec
       sbca #$01                         e6c3 e9 01     sbc #$01
       bne inner                         e6c5 d0 f8     bne $e6bf
       dex      ; all notes played?      e6c7 ca        dex
       bne outer ; no get next note      e6c8 d0 ed     bne $e6b7
       stx $e84a ; yes turn sound off    e6ca 8e 4a e8  stx $e84a ;x is double
                                         e6cd 8e 4b e8  stx $e84b ;one does al
       rts                               e6d0 60        rts
tabl fcb $00,$0e,$1e,$3e,$7e,$1e,$0e,$00 e74d 00 0e 1e 3e 7e 1e 0e 00
```

```
"chimer"
org $1000
"chimer.b09"
```

At left is chimer.cmd, needed for linking. Load chimer.mod in the monitor (not from menu!), then: >g 1000. The program will produce the chimes we are so used to hearing every time the 6502 side is called.

[Ed. You'll need good ears. We couldn't hear chimes until we put an ear against the case. The code is okay as is. Gary's second program, below, is easily heard. For those who want music: John Toebes of Raleigh claims that you can run both leads on the tiny speaker in SuperPET to a larger, external speaker and get good sound. We asked if more power was needed, and John said: 'No!']

The example above shows that translating 6502 code into 6809 is almost trivial. About 95% of the code may be translated directly. The major traps: both the X and Y registers are double-wide when compared to the equivalents in 6502; you must use finesse in handling them.

```
;sound1.asm
lda #$10
sta $e84b
lda #$0f
sta $e84a
lda # 00
```

Here is another example; it will cycle through all the tones available when using the CB2 sound of the the VIA (6522 chip) of SuperPET. (.cmd file, left.)

```
"sound1"
org $1000
```

As noted earlier, the 'nop' instruction produces a rather quiet sound; there is, however, a 6809 instruction

```
; main sound loop here
loop
  sta $e848
  nop
  inca
until eq
; turn sound off
lda #$00
sta $e84a
sta $e84b
swi
end
```

"sound1.b09" which causes enough delay to render a sound audible. I suggest you look for it (you'll learn a lot while looking); if nothing seems to work, see the end of this column.

Now that we've examined the rudiments of producing sound in 6809 assembler, perhaps you will want to try your hand at composing some simple songs. Try creating a table of notes for the X register and a table of delays for the Y register. The tabled references will let you play sequences in turn to create a song.

The next two articles will be devoted to two trouble spots: structured pro-programming statements, and determining the correct branch statement to use. The topics will be presented without quizzes. Note: Your 6809 delay is: SYNC.

◇◇

AN INDEX : COMING UP: PRINTING APL, TWO NEW DISKS, A STARTER-PAK This issue, we print an index of all Gazette issues from No. 2 through this one. It is not page-numbered, and can be removed from this issue if you want to file it in a 3-ring binder. We'll keep on printing Vol. I until we have enough pages to fill a 1.5-inch ring binder; then we'll start with Vol. II. Next issue, we have excellent material by Terry Peterson and Reg Beck on how to send the APL character set to MX80 and FX80 printers. AND: coming up, Terry Peterson's and Gary Ratliff's extended monitors on an ISPUG utility disk, together with a bunch of finished machine-language utility programs you can pick and choose from, AND a new 'Starter-Pak' which tells beginners how to start using SuperPET, with a tutorial disk (all the things the manuals do not cover, and which drove all of us up the wall when we first cranked up SuperPET). Both packages are just about finished; the Starter-Pak is designed especially for use in schools, and may be copied by schools for use by teachers and students. Please don't order either disk/manual until we announce they are ready.

◇◇

Prices, back copies, Vol. 1 (Postpaid), $ U.S.

No. 1: not available	No. 4: $1.25	No. 7: $2.50	No. 10: $2.50
No. 2: $1.25	No. 5: $1.25	No. 8: $2.50	No. 11: $3.50
No. 3: $1.25	No. 6: $3.75	No. 9: $2.75	

Send check to the Editor, PO Box 411, Hatteras, N.C. 27943. Add 30% to prices above to cover 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)

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. DUES ELSEWHERE: $25.00 U.S. Mail to: Paul V. Skipski, Secretary, ISPUG, 4782 Boston Post Road, Pelham, N.Y. 10803, USA.

Newsletter published by the International SuperPET Users Group (ISPUG); a non-profit association; purpose, interchange of useful data. Editorial offices at PO Box 411, Hatteras, N.C. 27943. Secretary, Paul V. Skipski, 4782 Boston Post Road, Pelham, N.Y. 10803. Membership applications, dues, and inquiries to Mr. Skipski; newsletter material to Hatteras, attn: Dick Barnes, Editor. SuperPET is a trademark of Commodore Business Machines, Inc.; WordPro a trademark of Professional Software, Inc. Enclose a self-addressed, postpaid envelope with all material submitted and all inquiries requiring reply. Membership: $15.00 per yr. U.S. in North America, $25.00 overseas and elsewhere. See enclosed application.

For all outside the U.S.: All nations members of the Postal Union offer certificates good in the postage of any other country for a small charge. The Union includes most nations of the world. Canadian members: send Canadian dimes or quarters for postage, but no paper currency.

FIRST CLASS MAIL

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

First-Class Mail
in U.S. and Canada