This is ISPUG's second anniversary, and we've come a long, long way--from the lonely days when each of us grappled in isolation with SuperPET, overwhelmed by what we did not know. Had it not been for ISPUG, the Editor knows he'd have given up on SuperPET. Defining the machine and its capabilities would have consumed forever had he been alone and without help. Your response to ISPUG and the Gazette has given each of us a broad view of the machine, has created SPET's splendid telecommunications capability (the programs now available either wouldn't exist or would be unknown), and has made the work of each of us available to all. We say to the members: "Well done!"

With this issue, we end Volume I of the SuperPET Gazette, and mark the occasion with the attached, cross-referenced index to that volume. Self-congratulation, however, won't improve ISPUG or the Gazette. And we do need to improve. Some ten percent of our members contribute all the material; ninety per cent giveth not and never write or call. We need your participation! Each of you can afford a few minutes every two months to tell us 1) what you'd like to know, 2) what you don't like about what we're doing, or 3) what you've discovered and think others might like to know. We'd like to get a postcard from every member every two months. Without your feedback, we simply don't know how well or badly we're doing and what we should print. Feedback, please.

Last, we need more articles and notes. They needn't be polished; the content only is important. The Gazette goes to hundreds of teachers, but we don't hear much about their problems. We'd like to. How else can we help? Is there a one of you who hasn't discovered something mighty useful about SuperPET? Why not tell us? While the Editor can't respond to every letter for lack of time, he'll try to, and appreciates every comment--pro or con, good or bad. We particularly want to see letters saying: "Hey, I'm stumped. Can you help out on..." If we don't know the answer, we'll try to find somebody who does. There's one caveat: we don't and won't cover the 8032 side of SuperPET, since **COMPUTE!**, the **MIDNITE REVIEW/PAPER**, and **MICRO** devote more space to the subject than ever we shall possess, while the 6809-side of SuperPET devours what room we have.

So, you silent ninety percent: off thy duffs and lend a hand. Write ye ed at PO Box 411, Hatteras, N.C. 27943 and tell him what you think of what we're doing, what to print next issue, or what you've discovered. ISPUG's charter is the interchange of information between members; you must give if you expect to get.

FAREWELL TO THE SECRETARY

Paul V. Skipski, our secretary, departs for distant fields and the university with this issue, and must resign his position. We thank him for much hard work, especially for publishing and distributing the Gazette. Henceforth, please send all inquiries and material concerning ISPUG to the Editor at PO Box 411, Hatteras, N.C. 27943. If any of you have ordered but not received disks from Paul, let ye Editor know, and we'll see you get 'em.

RED SEPTEMBER

We ran out of red ink whilst marking the mailing labels for this issue, since over half our members joined in September a year ago. Chances are better than even you'll find a note on your label that your membership has expired.

If so, check the RENEW block on the last page and send it with your address label or a copy (don't fill out the form if you send the label). Remit $15 U.S. in North America or $25 if elsewhere. And (sob!) please do it before we (sob!) must

scratch your name and address from the disk file and then (sob!) retype all that stuff when you finally do renew. Please take pity. Make checks to ISPUG.

ONCE OVER LIGHTLY
Miscellany If any of you wonder why your kids can't speak or write English, we suggest you see if their teachers can. We recently received a newsletter written by two Doctors of Education employed as Associate Professors of English. They tried to sell us a subscription (to teach us to write clear English)--in four pages of the most ponderous gobbledegook we've ever laid eyes on. As soon as they "heuristically mustered idea bases" to "reinforce a historical mission," we mailed the pompous asses a copy of Strunk on Style, with a note that if they'd read all 71 pages, both would drop dead from embarrassment. We'll print the next chapter on "The Murder of English by English Teachers" if we can remember where we stashed that fireproof paper.

ABOUT PRINTERS We get many inquiries on how to attach printers to SPET; not having the particular printer at hand, we can't reply. Yet we can pass along a few general tips. The IEEE port (dead center, rear of computer) is the place to start. Printers on the serial port must cope with SuperPET'S ACIA, and no word-processing program we've tried can do it. In addition, you have to switch, somehow, between printer and modem--if you have a modem--which'll force you either to buy a switch or to cable and uncable every time you go from printer to telecom mode. If possible, stay away from printers on the serial port.

Which leads to the question of where can you get an interface to convert from IEEE to parallel or serial? Well, a lot of new printers have an IEEE interface built-in. If you have a Commodore printer, you don't need an interface. If you have a parallel or serial printer, write or call Connecticut MicroComputer, 36 Del Mar Drive, Brookfield, Connecticut 06804, 203 775-4595. We've used their ADA 1450 for three years with no problems (IEEE to Serial). They make a number of interfaces for serial, parallel, etc. Write for the list. They provide an IEEE connector which plugs into the IEEE port on SPET and makes direct connection to your other IEEE devices. Piggbacked on top of the connector at the computer is their cabling to whatever interface you use. The cable connector to printer will vary, so know what rig you want at the printer end before you order.

If you should persist in using the serial port, be advised that QUILL Corp., 100 Schelter Road, PO Box 4700, Lincolnshire, Illinois 60198-4700, stocks a line of switches, including serial, most of which sell for less than $100--which is inexpensive as switches go these days. Yes, they sell cables/connectors too. Ask for their "Information Processing Products Catalog." Or call 312 634-4850.

TELL US SOONER, NOT LATER Gee, we got this nasty letter from a reader, which began with, "I sent you a change of address ONE WHOLE MONTH ago, and you sent the latest Gazette to my old address, which is my folk's home, and..." Sorry, but we make up the mailing labels ONE WHOLE MONTH before issues are mailed. In short, you gotta allow for some lead time. For this issue, which should appear in your mailbox about September 25, the copy deadline was August 15. We finished copy and mailed it to the printer on September 1--with a complete set of mailing labels. After that, the machinery runs all by itself, hands off. If your changes of address arrive after labels go to the printer, please see the agent in charge of the dead letter drop at Ultima Thule or write for another copy.

SAME DAY SERVICE EXCEPT ONCE IN A WHILE We always fill all incoming orders for Gazette back copies and ISPUG disks on the day they arrive. If you don't have the stuff you ordered in three weeks, including sortie time for the noble

employees of the Postal Service, scream at ye ed (who has goofed upon several memorable occasions) by letter or phone: 919 986 2443 Business, 919 986 2434 at home. Don't be bashful. Your Gazette is due on the 25th of every other month; if you don't have it by the tenth day of October, December, February, April, June, or August, scream again. We have known the the Postal Service to (ahem!) lose a copy now and then (an average of two copies every issue).

ANOTHER CRACK AT V1.1 We arranged a few months back to supply V1.1 languages to ISPUGgers who couldn't get them, thinking that would take care of everybody. But (sob!) a lot of new people joined and want V1.1 disks plus updated pages for their manuals. So, once more into the breach, dear friends, for Harry, St. George, and England--order V1.1 now, please. We'll consolidate the order and again order a bunch, which we cannot deliver in less than one month to six weeks. Send $32.95 U.S. for 8050, or $38.95 for 4040 format. We'll send back the disk (disks in 4040) plus manual update pages plus COBOL manuals. Anybody using mBASIC, mAPL or mCOBOL will find V1.1 essential; mFORTRANners and mPASCALiers can squeeze by with V1.0, but we suspect the ability to send directories to disk or printer in V1.1 is alone worth the cost. If you're gonna order, do it now! Price includes postage and handling. Order from Editor, PO Box 411, Hatteras, N.C. 27943.

PIDGIN To save space, we'll henceforth refer to all Gazette articles by Volume and page, as in I, 213--which means Volume I, page 213. When you see such cryptic notes in the index to Volume I, we hope you remember what they mean.

TELECOM WITH THE 2031 DRIVE We recently printed a note that you might drop received characters if you use a 2031 drive with NEWTERM or FASTERM. We just had a report from Barry Bogart that he is able to use COM-MASTER (I, 136, 210, 222), with his 2031 with no dropped characters. This again confirms our view that COM-MASTER is a first-rate terminal emulator.

ISPUG DISKS We gotta great buy on 3-M disks (haven't lost a one in three years). If you want 50 or more, order from Disk World, Suite 4806, 30 East Huron Street, Chicago, Illinois 60611 (call 1-800-621-6827 anywhere except Illinois, which is 1-312-944-2788). Visa or Mastercard. $1.60 per disk, single-sided and double density, with Tyvek envelopes, write-protect tabs, reinforced hub, but no box. If you get ISPUG disks with no maker's label, you know you have 3-M, not some cheap junk. Yeah, Disk World sells in small quantities too. Call.

FAILURE POINT This is the second report of the problem, so be aware of it. The SPET owned by John C. Wyatt of Lexington, Kentucky failed; he found the 8-wire lead from transformer to the lower board had failed at the red connector, by burning the insulation off the white insulator, female side. After he cleaned the connections, replaced insulation, and increased connector spring tension, SPET revived. Same problem, same cure from John Thomas of Pittsburgh.

VISICALC OK IN THE BANKS John Wyatt also reports he's loaded ROM images of the ROMs which fit at U45 on new, two-board SPETs (address $9000-$9FFF) or at UD12 on old, three-board machines (same address). For how to do this, see Roy Busdiecker, I, 28 ff. John reports Visicalc works fine when loaded in any of the switched banks in 6502 mode using Roy's method.

THE MONITOR FROM EVERY LANGUAGE Russ McMillan of Madison, Wisconsin reports that QUAD SYS 61631 gets the monitor in APL. Well, shucks, a SYS 61631, appropriately sent in any language, hits the monitor entry point, and a 'q' in monitor returns you to mBASIC and APL (see Gary Ratliff in I, 227). What you see in the

monitor, thus SYS'd, is different from the report you get when you enter the monitor from mED in that language. Unfortunately, the return address for the SYS call is missing in mFORTRAN and mPASCAL; you can't return to those languages.

ZELLER IS MISSING part of his computer again. No column this issue. Sorry.

HIGH RES GRAPHICS or COMPUTER-ASSISTED DRAFTING If anybody has a program to provide either capability in SPET, either in 6809 or 6502, please let us know. And write Shawne Ross, 3029 Keighly Road, Nainamo, B.C. Canada V9T 2X9. She's aware of 6502 packages from High Res Technologies in Toronto, and from Automated Design of Valley Forge PA, but has no info on how well they work. Help!

LET'S SEE SOME SORTS! One of the most widely useful of all utility routines is the string sort. Note we didn't say integer sort! A string sort will handle integers, but it doesn't work vice versa. We publish below a modified shell sort to get this series going (we hope). Well-structured, it should be useful in all languages. If you have a fast string sort in any language, send it in!

We thought that transferring strings in an array as the sort proceeded was a lot slower than a simple switch of integer subscripts in a sorting array--but found, to our surprise, that 'twas not so. A version using subscript switching ran 18% slower on long lists than the string-transfer version below. Only when strings are much longer than ordinary English names does a subscript-switch run faster in microBASIC. Feed the sort below with a single string per disk input line.

```
100 option base 1 : one%=1 : two%=2 : zero%=0 : CS$=chr$(12)
105 ! num_items% is number of items to be sorted.
110 ! list$(n%) is a specific line entry to be sorted in matrix list$.
...
200 print CS$; 'Sorting...'
205 half% = num_items%/two% : halflist% = num_items%-half%
210 for j% = one% to num_items%
215   for i% = one% to halflist%
220     if list$(i%) >= list$(i%+half%)
225       transfer$ = list$(i%) : list$(i%) = list$(i%+half%)
230       list$(i%+half%) = transfer$ : exchange% = i%
235     endif
240    next i%
245    if exchange%
250      halflist% = exchange%-one% : exchange%=zero%
255    else
260      half% = half%/two% : halflist% = num_items%-half%
265    endif
270    if half% < one% then quit
275 next j%
280 print 'End of sort'
```

At end of sort, the matrix list$ holds the strings in alphabetical order. The sort itself separates capital and lower-case letters. If you want capitals and lower case in the same order, knock the capitals to lower case (or vice versa) either when you create list$, or as you pull the strings off disk to be sorted. When sorting names, it's easiest to knock all initial letters to lower case as the string comes off disk, mark the string with tilde or other non-used charac-

ter, sort, and then reconvert the string as filed to disk. If 'tweren't for such prefixes as "le" and "l'", and names such as MacDonald, sorts would be simple.

TO SENSE, DEFINE, AND REDEFINE THE KEYBOARD Some time back, we got from Paul Ruud of Berkeley, CA, an assembly language program which redefined the keyboard. Later, Paul demonstrated the program to Terry Peterson, who adapted it to HELLO/GSCROLL, a program issued on the ISPUG Utility disk which redefines the keyboard. Unfortunately, neither Paul nor Terry explained what he had done or how he had done it. When finally we got the time, we dissected the method. It's powerful and useful for anyone who uses interrupt-driven programs or who wants to reconfigure the keyboard.

Let's start with the way SuperPET senses a keypress. Every 1/60th of a second, the normal IRQ interrupt routine checks for a depressed key. If one is sensed, the key value is translated into SuperPET's usual ASCII code sequence. The value which comes back from the keyboard is stored in one byte at $012B, which we call KyIndx_. The value in KyIndx_ is an offset into a ROM table at $DF40, for at that offset you'll find the SPET code for the key. Here's an example: Press the HOME key; the value in KyIndx_ at $012B is $0C (decimal 12). If you offset 12 bytes into the table at $DF40, and lock at $DF4C, you find (surprise, surprise) that the value assigned to HOME is 1, which indeed is the code we expect. All offsets found in KyIndx_ at $012B work exactly the same way.

Isn't it immediately obvious that if you can define the keyboard from a table at $DF40, you can redefine it from your own table in any assembly-language routine? At left, below, is a simple example. First, we sense any keypress with system routine KYPUTB_ at $DD82. Suppose we press the STOP key. From the offset table at the end of this article, we find that STOP will return an offset of $04 in KyIndx_. The program at left offsets 4 bytes into TABLE by adding, in the last line, the offset of 4 bytes to the address of TABLE, and then loads the value of $34 from TABLE into B register (yes, you can load into the same register you use for an offset). Note that though you press STOP, you return a decimal 4 in B register ($34 is ASCII for "4"). The method is easy, painless, and fast.

```
loop
  jsr kyputb_
until ne
ldx   #TABLE
ldb   kyindx_
ldb   b,x

TABLE  fcb 00,$31,$32,$33,$34,$35
;bytes      0   1   2   3   4   5
```

Next, if you want to write an interrupt-driven routine which will not commence with any usual combination of keys, you can assign as "triggers" some key combinations SuperPET doesn't use. SHIFT/LEFT ARROW ("_"), for example, has its own offset code. In the ROM table at $DF40, both shifted and unshifted offsets for "_" are translated to "_", though those offsets are totally different ($08 and $59). Similarly, RETURN and SHIFT/RETURN generate two distinct codes; the "2" key on the main keyboard returns a code different from Keypad "2", and SHIFTED Keypad "2" yet another offset. In short, each key returns an utterly unique offset, which you may sense at KyIndx_ and employ as you wish. At left, we use SHIFT LEFT ARROW, offset $59, to kick off an interrupt-driven routine. The code at left will be scanned at every IRQ, each 1/60th of a second. If SHIFT and LEFT ARROW are depressed, we execute that routine. See separate article, this issue, on the general approach to handling interrupt routines. Save the offset table below; it is handy and valuable.

```
; Trigger for
; interrupt routine
ldb   kyindx_
cmpb  #$59
if eq
   enter routine
```

You'll find a table supplement starting at $DFE2, which is used to define the APL keyboard. The APL supplement does not work on direct offsets of KyIndx_, but only with the system ROM code at $DEF3.

We try to anticipate some confusion on key codes with the following explanation of what seems to go on:

1. The PIA at $E810+ senses a keypress with its own coding system ($FF is no key down). The system IRQ routine senses what goes on in the PIA, and converts the PIA keypress value either to a Waterloo Roman Value or to a Waterloo APL offset into the key table at $DF40 forward.

2. The value of the offset is stored in KyIndx_; when the proper keyboard is sensed (APL or Roman), the index is converted to an ASCII or APL key value. This key value is placed in the keyboard buffer and KyPtr1_ and KyPtr2_ are adjusted to show the location, in the buffer, of this character.

3. In short, you will find at least three different values for any key: the PIA value, the value in KyIndx_, and the ASCII or APL table value. All three are useful. The table below shows the KyIndx_ value for the ASCII keyboard.

ASCII KEYBOARD TABLE OFFSETS IN ASCII CODE ORDER

Values Taken from Table, $DF40 Forward Current Key Offset Stored at $012B

ASCII Decimal		Char/ or Key	Offset (Hex)	ASCII Decimal	Char/ or Key	Offset (Hex)	ASCII Decimal	Char/ or Key	Offset (Hex)
00	NUL		00	53	5	4F	94	↑	43
01	SOH	Home	0C	54	6	06	95	←	08
02	STX	Shift/Run	55	55	7	46	95	Shift/←	59
03	ETX	Stop	04	56	8	4E	96	` (Shift/@)	83
04	EDT	Delete	29	57	9	05	97	a	38
05	ENQ	Shift/Insert	7A	58	:	03	98	b	1E
06	ACK	Escape	40	59	;	3A	99	c	1F
06		Shift/Escape	91	60	<	66	100	d	37
07	BEL	Cursor Right	4B	61	=	9E	101	e	27
08	BS	Cursor Left	9C	62	>	6E	102	f	3E
09	HT	Tab	30	63	?	5B	103	g	36
09		Shift/Tab	81	64	@	32	104	h	3D
10	LF	Cursor Down	24	65	A	89	105	i	2B
11	VT	Cursor Up	75	66	B	6F	106	j	35
12	FF	Shift/Clear	5D	67	C	70	107	k	3B
13	CR	Return	34	68	D	88	108	l	33
13		Shift/Return	85	69	E	78	109	m	0D
14-31		.....Deleted.....		70	F	8F	110	n	16
32		Space	0E	71	G	87	111	o	23
33		!	99	72	H	8E	112	p	2A
34		"	A1	73	I	7C	113	q	28
35		#	58	74	J	86	114	r	2E
36		$	98	75	K	8C	115	s	3F
37		%	A0	76	L	84	116	t	26
38		&	57	77	M	5E	117	u	25
39		'	97	78	N	67	118	v	17
40		(	9F	79	O	74	119	w	2f

41	)	56	80	P	7B	120	x	0F
42	*	54	81	Q	79	121	y	2D
43	+	8B	82	R	7F	122	z	18
44	,	15	83	S	90	123	{	73
45	-	4D	84	T	77	124	¦	7D
46	.	1D	85	U	76	125	}	8D
47	/	0A	86	V	68	126	~	94
[Numb. below main keyboard]			87	W	80	127	Repeat (DEL)	12
48	0	45	88	X	60		Shift/Repeat	63
48	Shift 0	96	89	Y	7E		Reverse	10
49	1	48	90	Z	69		Shift/Off	61
50	2	50	91	[	22	All shifted keys have		
51	3	07	92	\	2C	different values than		
52	4	47	93	]	3C	unshifted.		

	Unshifted Keypad				Shifted Keypad		
Char.	Hex Offset	Char.	Hex Offset	Char.	Hex Offset	Char.	Hex Offset
.	1C	4	21	.	6D	4	72
0	14	5	39	0	65	5	8A
1	09	6	31	1	5A	6	82
2	11	7	44	2	62	7	95
3	19	8	4C	3	6A	8	9D
		9	41			9	92

TWO FUNCTION DUMPS In APL, it'd be handy to dump an entire function to printer or disk, especially when a function is too long for the screen. Reg Beck of Williams Lake, B.C. sent one in (below, left) for use with his FX80 printer. At the same time, Russ McMillan of Madison, Wisconsin, told us about a problem with his Okidata printer--which won't backspace. What do you do about overstruck characters with such a beast? Russ sent in a function which requires the printer to make two passes. We sent him a copy of Beck's function dump, which he adapted to the Okidata. Since our membership applications show a lot of Okidata printers out there, we print Russ's function dump for the Okidata below. It sends to printer any function in WS, no matter how long it is. To use either of these functions, give commands in this way: FDUMP 'FDUMP', which causes FDUMP at the left to dump itself to an IEEE4 printer (mit line numbers yet!).

```
∇FDUMP[□]
[  0]    FDUMP F ;I;HDR;N
[  1]   HDR←F
[  2]   F←□CR F
[  3]   N←1↑ρF
[  4]   F←' ',F
[  5]   N←0 ¯1↓∇(N,1)ρ¯1+ιN
[  6]   F←('[',N,']'),F
[  7]   'IEEE4'□CREATE 1
[  8]   ('    ∇',HDR,'[□]∇')□PUT 1
[  9]   I←0
[ 10]   LOOP:   I←I+1
[ 11]           (□XR F[I;])□PUT 1
[ 12]   ENDLOOP: →(I<1↑ρF)/LOOP
[ 13]   □UNTIE 1
```

```
∇FOKIDUMP[□]∇
[  0]    FOKIDUMP F ;I;HDR;N;□IO;SCR;X;Y;W1;W2
[  1]   HDR←F
[  2]   □IO←0
[  3]   F←□CR F
[  4]   N←1↑ρF
[  5]   F←((N,1)ρ' '),F
[  6]   N←0 ¯1↓∇(N,1)ριN
```

```
[  7]   F←('[',N,']'),F
[  8]   'IEEE4' ⎕CREATE 1
[  9]   (⎕XR,'     ∇',HDR,'[⎕]∇') ⎕PUT 1
[ 10]   I←0
[ 11]   LOOP: SCR←⍳0
[ 12]   SCR←⎕XR,W1←W2←,F[;]
[ 13]   Y←¯1+X-2×⍳ρX←(SCR=⎕AV[8])/⍳ρSCR
[ 14]   W1[Y]←SCR[¯1+X]
[ 15]   W2[Y]←SCR[ 1+X]
[ 16]   ((W1,'⍝=⊥0001'),W2) ⎕PUT 1
[ 17]   I←I+1
[ 18]   ENDLOOP: →(I<1↑ρF)/LOOP
[ 19]   ⎕UNTIE 1
```

TWO LESSONS IN SYSTEM CONTROL In both mFORTRAN and mPASCAL, program results disappear when a program's finished and when you leave the debuggers. When debugging, you often want a record of what happened. To use a machine-language dump you must STOP the program or be mighty nimble in keying its start. Sometimes you simply want a record of what a program does as you modify and develop it. For these reasons, we wrote an mFORTRAN dump which you may 1) call within program to send the screen to printer, or 2) call in debugger mode with: e call dump <RETURN>. In either event, it sends up to 23 lines to printer and then stops. If there are fewer lines than that, it stops itself on the last line of program output. Make it a dump to disk simply by opening a disk file instead of a printer file. But the lessons lie in the control achieved and not in the dumps per se.

Unfortunately, the mFORTRAN cursor destroys screen characters when you print or write. To avoid this, we move the cursor with TPUTCURS_, and input with the cursor on column 80. Originally, we dumped by stuffing a CR into the keyboard buffer and asking for a READ--which automatically dumps the buffer, executes the READ, and assigns what is on a line to the character variable so read. But we couldn't READ anything past the first comma !#$*!. We asked Stan Brockman if he could whip that one, and he most ingeniously did, by 1) using system procedure GETREC_ to read a record from the screen, and, 2) finding the string address by use of intrinsic function VARPTR, often dangerous, but here a charm. Thanks to Stan, we now have a system technique which should work widely and well.

At the same time, we asked grand sachem Terry Peterson how he'd solve the problem, and back came a totally different approach which uses FORMAT statements to do the job. Terry's approach depends only on the built-in functions of FORTRAN, and is system-independent (though you might CONTINUE if you can't ENDDO).

```
write(6,rec=0) char(12)//rpt('x,',38)//char(13)  * This preface forms material
do i=1,20                                        * with gobs of commas, a blank line,
  print,cnvi2c(i)//rpt('.,',38)                  * more characters and commas,
  if (i=10) then print,'xxx'                     * a short line,
enddo
print,rpt('x',30)                                * and a medium line. All record properly.
call dump
end                                              * Preface is purely for demonstration.
```

```
subroutine dump                          * STAN BROCKMAN'S SOLUTION:
  integer tputcurs,tgetcurs,getrec
  character line

  open(10, file='ieee4')
  open(20, file='keyboard')              * We stuff a carriage return in here later,
  tputcurs=cnvh2i('b087')                * move the cursor with this system routine,
  tgetcurs=cnvh2i('b084')                * and sense cursor position with another.
  getrec=cnvh2i('b0c3')                  * Gets record from terminal (p. 176, Assembler).
  istop=sys(tgetcurs)/256                * What row is cursor on when program stops?
  line=rpt(' ',80)                       * Set up a dummy line for VARPTR to find.
  iaddr=varptr(line)                     * What's the address of the line in mFORTRAN?
  j=sys(tputcurs,256+79)                 * Put cursor on line 1, column 79. Why 79? We
                                         * don't know, but 80 fails to dump two lines.
  do i=2,25                              * The 'i' is the screen line.
    write(20) char(13)                   * Stuff a CR in the keyboard buffer, and dump it
    ilen=sys(getrec,iaddr,79)            * when GETREC wants a record from the terminal.
    if (ilen.ge.1) then                  * GETREC stores it, returns 'ilen'=record length.
      write(10) line(:ilen)              * So write it to printer, for the length known.
    else
      write(10)                          * Or if it's null, print a blank line.
    endif
    quitif i >= istop                    * Are we down to end of program yet?
10  x=sys(tputcurs,(i*256)+79)           * Put cursor at end of next line to be dumped.
  enddo          [Comments: Does division by 256 on ISTOP and multiplication by
  close(10)      256 on line 10 seem redundant? Remove the "/256" from ISTOP and
  close(20)      change DO to i=256,istop,256. Let line 10 be...(tputcurs,(i+80));
end              then try (tputcurs,(i+79)). THEN quietly go mad. What happens?]
```

```
subroutine dump                          * TERRY PETERSON'S SOLUTION
  character line

  print, "QUIT"//char(11)
  open(10, file='ieee4')
  open(20, file='keyboard')
  open(30, file='(f:80)terminal')

  do i=1,24
    write(20,"a1") char(13)
    read(30,rec=i-1,fmt="a79") line
    quitif line='QUIT'
    write(10) line
  enddo
  close(10)
  close(20)
  close(30)
end
```

The formatted WRITE and READ in the program at left get a line of 79 characters without regard to commas. If you attempt to get 80 characters, the material will double space to disk or printer.

Both subroutines depend on a CR dump. All READ, INPUT, LINPUT, STOP, END, and GET routines we've tried in all languages look to the keyboard buffer for operator input and/or a CR. If a CR resides there, it is accepted, along with the input line which the CR confirms as if you'd entered it.

Note how Terry takes advantage of SuperPET's ability to use the SCREEN as a relative file (V.I, p. 134). If you're troubled by the recurring value of 256 in Stan's program, see I, pp 213-216, issue 13.

Next issue, we'll show how to do the same thing with an ML dump, written into microFORTRAN and using position-independent assembly language code (PIC).

WHAT'S AN INTERRUPT?

by Loch Rose
Associate Editor
102 Fresh Pond Pkway
Cambridge, MA 02138

Now that interrupt-driven routines are popping up all over the Gazette, a lot of people must be asking this question. An interrupt is a signal originating outside the 6809 microprocessor (I'll call it the CPU) which causes the CPU to stop whatever it is doing and start processing an interrupt-handling routine instead. This discussion will apply to the 6809 CPU used by the Waterloo languages, which is capable of responding to several different kinds of interrupts (of which, more later). The discussion below covers only regular interrupts (IRQs).

There are three things that you should understand about interrupts: first, before the CPU responds to an interrupt, it saves a complete record of its current state. When it returns from processing the interrupt, it continues where it left off, exactly as if nothing had happened. Second, it sets a flag that causes the CPU to ignore any further interrupts while it is dealing with this one. And, third, the address of the interrupt-handling routine is stored in RAM, so you can alter this address and cause the CPU to execute your own interrupt routine instead of, or in addition to, the one supplied by Waterloo in ROM.

SuperPET has a clock which sends an interrupt signal to the CPU every 1/60th of a second. SPET thus executes an IRQ interrupt-handling routine 60 times each second. Because of a lengthy investigation by Joe Bostic, Terry Peterson, and Dick Barnes (Ed. And by Loch Rose!), we know how this routine does three things: 1) increments the system clock by 1/60th of a second and resets seconds, minutes and hours in the process; 2) checks the keyboard to see what key, if any, is now depressed, and, 3), puts the character corresponding to that key (ASCII or APL) into the keyboard buffer, and at end of interrupt, puts the character to screen.

If you make the CPU execute your own interrupt-handling routine, that routine should normally call Waterloo's routine as well, so that the system clock is updated and the keyboard continues to work. What else might your routine do? You will find interrupt-driven routines useful when you would like the processor to check for some event, yet want to employ the computer normally at the same time. For example, you could write an "alarm clock" interrupt routine; you could specify a time; at every interrupt, the CPU would check the system clock to see if the time had been reached; if so, you could print a message, require the speaker to sound an alarm, etc. In the meantime, you'd use the computer normally, never seeing any evidence that the routine checks the clock every 1/60th of a second.

Other examples are the screen dumps to printer on interrupt (PIRQ and UDUMP), described in I, 73 and I, 166. PIRQ becomes active only when it finds 40 backslashes in the keyboard buffer; UDUMP comes alive only if you press PF7. Any program in language or any system facility (such as the mED) meanwhile continues to run normally before and after the interrupt.

Interrupt-driven telecommunications programs (BASICOM or COM-MASTER, for example) also rely on user-written interrupt-handling routines, in which you instruct the ACIA (the chip which runs the serial port) to create an interrupt whenever it receives a character from the serial port. At interrupt, therefore, you check the ACIA to see if it needs service. If it does, you must fetch a character from it before that character is overwritten by the next character to arrive at the serial port. If the ACIA didn't interrupt, you instead perform only normal IRQ service. Such an approach makes possible very high speed serial communications.

I hope this preface gives you an understanding of the following material, consolidated and written by Dick Barnes, on how to handle user interrupt routines.

INTO THE WILDERNESS WITH THE EVIL DR. MORIARTY
and a Glimpse of the Promised Land

The wilderness is not that of Sinai, but of interrupts; and, to mix a metaphor, we tell a detective story filled with uncertain clues, a horde of confused detectives, and a solution which will only whet the appetite of readers for a better ending. While you may be confused by side issues (many a bush will burn), we do promise to get both feet into the Promised Land--i.e., to deliver two working solutions to the main problems you face in employing user interrupt routines.

Thanks go to Terry Peterson, Loch Rose, and Joe Bostic for very active and most helpful detective work; for many a clue and for the excellent suggestions which emerged from their endeavors.

There are a number of different types of interrupts: fast (FIRQ), regular (IRQ), software (SWI, SWI2, SWI3), Non-Maskable (NMI), etc. We concern ourselves here only with regular or IRQ interrupts. When we use the word "interrupt" hereafter without qualification, that's what we mean.

Each of the interrupt types is serviced or handled in a different way. Waterloo provides a table at $0100, called IntVctr_, which holds the addresses of the routines in ROM. We dumped the table in the monitor in mED and commented it (see below). As an example, note that a regular IRQ interrupt is serviced by a routine which starts at $DE0B in ROM (second line).

Contents of Table at $0100, Soft Copy of Interrupt Vectors, in mED.

Type Interrupt:	Reserved		SWI3		SWI2		FIRQ	
Address $0100+	0	1	2	3	4	5	6	7
Vector:	ff	b1	ff	b1	ff	b1	ff	b1

Type Interrupt:	IRQ		SWI		NMI	
Address $0100+	8	9	a	b	c	d
Vector:	de	ob	fd	de	fd	de

Now open the assembler manual to pp. 187-188. You'll find a system routine called ConBInt_ uses some numbered parameters to identify what sort of interrupt is to be serviced. An IRQ interrupt has the parameter of 8. The table above shows that an IRQ is serviced by the routine whose address is at $0108-$0109. Yup. The numbered parameters for ConBInt_ are simply offsets from start-of-table, $0100.

What does ConBInt_ do? Well it's designed to "Connect Bank Interrupts" to user routines in a switched bank. Waterloo didn't tell us that there's another routine, ConInt_, which connects user routines for programs in low memory (not in the switched banks). We didn't know about it until Joe Bostic pointed it out, at $FF97. You can use ConBInt_ to connect your routines to regular routines okay, but the code is a lot longer and more complex than you need if you're not working in the banks. Use either routine to hook your routine to the regular one.

When you set up your own IRQ routine, you need to do two things: (1) tell SPET to provide normal IRQ service and to service your routine, and (2) write your own IRQ routine. Let's take the steps one at a time; first, what we do to get normal IRQ service plus service of our routine:

```
ldx    #8           ; Load the table offset of 8 bytes into X register.
ldd    intvctr_,x   ; Offset 8 into the table at $0100, and get the address of
                    ; the IRQ handling routine, as shown above.
std    normserv     ; Save the address so we can call for normal IRQ service.
```

Next, we have to insert our own routine at an IRQ interrupt. We use ConInt_, which takes two parameters; P2 going onto the stack, and P1 in the D Register.

```
pshs   x            ; P2, the type of interrupt to be serviced (8, above, an IRQ)
ldd    #ourtask     ; Load the starting address of our interrupt-driven routine.
                    ; This is Parm 1 for system routine ConInt_.
jsr    conint_      ; So, add our routine to the normal routine.
```

Let's put Step 1 together in a code package called MAIN. We'll go into the details of the system routines and of the pointers at the left when we reach Step 2 and write our routine. The only feature not discussed above which has been added to this code is the set of MemEnd_ to keep our module from being overwritten by other programs. (We suggest $7000 as origin for this program. Load it from main menu; kick it off with PF7 anywhere.)

```
xref  printf_
memend_   equ $22
intvctr_  equ $0100
kyindx_   equ $012b
conint_   equ $ff97
kyptr1_   equ $012C

main      equ   *
          ldd   #main
          std   memend_
          ldx   #8
          ldd   intvctr_,x
          std   normserv
          pshs  x
          ldd   #ourtask
          jsr   conint_
          leas  2,s
          clr   $32
          rts
ourtask   jsr   [normserv]
          ldb   kyindx_
          cmpb  #$95
          if    eq
            ldd #message
            jsr printf_
          endif
          rts
message   fcc "We pressed SHIFT 7%n"
          fcb 0
normserv  rmb 2
          end
```

When our program loads at main menu, it automatically stops at the RTS at left. The next part of our program (which we'll cover in Step 2) will now be checked at each IRQ to see if a specified "trigger" condition instructs it to run; or, if it's a general-purpose routine with no trigger, it will execute at every IRQ. We now pass to Step 2, and write the routine itself--called OURTASK.

First, we must arrange for normal IRQ service with the JSR to "normserv". Then we add a short routine which checks to see if PF7 has been pressed; if it has, we print a message. The routine works after a fashion, but mostly reveals the problems we face. Strongly suggest you enter, assemble, and link it so you understand the problems and their solutions. See separate article, this issue, on KyIndx_, which we load to find out if PF7 has been pressed. If it has, we print a message to screen at each IRQ. You will best see the problems if you press PF7 at menu and in monitor: 1) The message "We pressed SHIFT 7" will print several times for each keypress; 2) the menu loader will think you want to load PF7's graphics character from disk one and will pour out "Program Not Found" messages, 3) if you're in the monitor, the graphics generated by PF7 will create "Invalid Command" messages, and, 4) the message, "We pressed SHIFT 7" will be split into two or more parts. Most obviously, our interrupt routine is being interrupted during an interrupt! Oh, woe. What do we do?

Let's tackle each of the problems set forth above, one at a time. Then we'll put

a routine together from the various solutions. First, let's look at that strange interrupt:

Moriarty's Interrupt The evil Dr. Moriarty, infamous opponent of Sherlock Holmes, apparently has taken up residence at Waterloo. Although we first thought that some system routines, such as PRINTF_, cleared the Interrupt Flag (bit 4 in the CC register), and thus allowed another normal IRQ interrupt to occur right in the middle of a user routine, Loch Rose and Joe Bostic later discovered that part of the IRQ routine at $DDAD sometimes calls a subroutine at $E765--and that routine clears the I flag! Which means, of course, that you can pass from a normal IRQ service routine to your routine with the I flag set to zero, so that normal interrupts can resume right spang dab at the start of your user routine. Terry P. demonstrated this with the code at left, above, which senses the I flag in the CC register and forces any subsequent IRQ to bypass user code. We still think it utterly mad that Waterloo should allow Moriarty to clear the I flag as you enter a user routine, but that's the way it is. How do we stop this nonsense? Terry shows one way, above.

```
tfr   cc,a
anda  #%00010000
if    ne
 ...enter routine
```

The Busy Flag We found another answer in one of Gary Ratliff's old routines: set a Busy Flag in our routine whenever it is being executed, and clear the flag only when our routine has ended. If Moriarty allows his interrupt, this method shunts Moriarty's call around our routine, which doesn't execute. This technique we employed in UDUMP (I, p. 167, No. 11); that dump is most reliable, so we know the method works. The code for it is at left. There is a third way which is simplest and which has been successful. You set the I flag with SEI at the start of your routine. This reset of the Interrupt Flag keep Moriarty out. Both the Busy Flag and SEI approaches prevent double execution of a user routine. The simplest of the three methods (include Terry's, above) is SEI, but it can create two problems: 1) if you need to sense a keypress, you must resort to an unusual method, and, 2) by shutting off interrupts, you prevent normal system services (such as updating the clock, at $015E upward). If your routine is short and you don't mind the clock getting a bit off, SEI works well.

```
tst busy
if eq
   dec busy
   ...rest of routine
   clr busy
endif
```

Repeated Messages Few of us have the delicate touch to hit PF7 and release it in 1/60th of a second or less, so a number of IRQs sense the key down. Each IRQ prints our message. We can avoid this four ways: 1) stuff in a simple time delay (see that at left, which increments 64K before it quits). Stick that in our routine, and we print once for each keypress unless we put our foot on PF7; 2) write a long routine, which in execution effectively creates a time delay, 3) set up a routine which pauses for keyboard input--and also creates a delay. The fourth method, again thanks to Joe Bostic, won't proceed until a key is released. At left, we sense the keyboard with a system routine at $DEF3 (KYSCAN_), which does not call that nasty code $E765, and does not clear the Interrupt Flag. Our program will not proceed until the last, sticky finger is removed from the keyboard. This is a neat, simple approach. See below.

```
ldx  #0
loop ;Delay
  leax 1,x
until eq

;Key Release
loop
  jsr $DEF3
  tstb
until eq
```

To Sense Keys in an Interrupt Routine This is conceptually tricky: if IRQs are masked off, how do we sense any keypress after we're in an interrupt rou-

tine? At the start, a normal IRQ service routine scans the keyboard, but how can it rescan without another IRQ? This turns out to be far less difficult than it might appear. We have two ways to go about it. In the first, if we want to have full access to the whole keyboard, we can use system routines such as GetChar_ or KyPutB_, but we must clear the Interrupt Flag to use them. If we do that, we must also use the Busy Flag technique, so that subsequent interrupts don't run us through our routine umpteen times. The code to do this is shown at the left; our first program solution, at the end of this article, uses the Busy Flag technique with CLI. It works very reliably.

```
cli
loop
  jsr kyputb_
until ne
```

The **second way** to sense keys during an interrupt routine emerged from a suggestion by Joe Bostic that we use the regular IRQ keypress routine at $DEF3 (KyScan_); it works splendidly if you call the routine three times. The first call must be to wait until the trigger key for the routine is released. You JSR to KYSCAN_ at $DEF3 until the return is 0. Then you call the routine again, as at the left, below, until a key value returns. And then (ho hum), you call the routine for a third time to wait until the key you just pressed has been released! We put this trio of calls together in one of the sample routines which follows, so don't worry about the details until you see how it can be done. There's yet another way to get a keypress, using the PIA at $E810 forward; we'll cover that in the next issue of the Gazette.

```
loop
  jsr $DEF3
  tstb
until ne
```

Disk/1 Loading Messages and Invalid Command Errors in Monitor Each IRQ scans the keyboard for a keypress; at the end of IRQ, the character for the key prints to the screen. In our first routine, shouldn't we expect the character for PF7 to so print? Of course! Our problem is to get rid of it. You can do this with a club by clearing the whole keyboard buffer (see comments on how on page 168, issue 11), but there's a neat, quick, surgical way to delete only the character in question. There are two pointers to the keyboard buffer; they index the position in the buffer of the last character entered from the keyboard. You will find the pointers at $012C and $012E. We use the first, called KYPTR1_, and simply tell SPET to clear that location. Voila! The character is deleted; we get no more error messages at menu or in the monitor. The method clears any unwanted character in the keyboard buffer.

```
clr  [kyptr1_]
```

To Leave an IRQ Routine and Reset to Normal Service Suppose you want to leave your routine and cancel IRQ service for it. You simply load the normal address for regular IRQ service and stuff it into the table at $0108. The code to do it is at left. Of course, you don't do this while a BUSY flag is set, or you'll cancel your routine when it hasn't been finished. In the programs which summarize this article, we use SHIFT LEFT ARROW to disconnect from our user routine (see separate article, this issue, on the codes for the various keys). You can use ConInt_ to do this, but it's not necessary. ConBInt_, however, probably should be used to disconnect if you connected your routine with it at the beginning. The code for a disconnect from your routine, using ConBInt_, is shown at the left. Note the length of the code, compared to that above.

```
ldd  normserv
std  invctr_ + 8
```

```
ldd  #8
pshs d
ldd  normserv
jsr  conbint_
leas 2,s
```

We put all the above together in two demo routines which follow. Be sure to include MAIN (at start of article) to begin your program. The routines below are the second part; they replace the short but messy OURTASK printed at the start.

The approaches show two (of many) ways to get a working routine with every one of the features we've discussed, for we: 1) Trigger a user routine with PF7, 2) print a message, 3) ask the user to press a key, 4) pause whilst doing so, 5) give the user the option to cancel the routine with SHIFT LEFT ARROW, and, 6) tell the user where he is in the program at all times (and when he cancels it).

A word of warning from Joe Bostic: don't try to return from your own IRQ routine with RTI (Return from Interrupt); the system routine we hitch onto does this for you. If you RTI, thou crashest. Instead, use an RTS to get back into your regular program, as we do in the examples below.

Last, throw away your ordinary concepts of time. Without the pauses for key release or the shunt by the Busy Flag, all actions occur in a few millionths of a second. You must slow the process down to fit human reaction time.

THE BUSY FLAG APPROACH

```
ourtask jsr   [normserv]     ; Get normal IRQ service.
        ldb   kyindx_        ; Load offset value of last key pressed.
        cmpb  #$59           ; Is it SHIFT LEFT-ARROW?
        if eq
          clr   [kyptr1_]    ; Get rid of SHIFT LEFT ARROW character.
          tst   busy         ; Don't clear routine if we're in it!
          bne   fini
          ldd   normserv     ; If not, clear interrupt routine and resume
          std   intvctr_+8   ; normal IRQ service.
          ldd   #discon
          jsr   printf_      ; Tell user he's disconnected from his IRQ routine.
          bra   fini
        endif
        cmpb  #$95           ; Is last key PF7?
        bne   fini           ; If not, quit.
        clr   [kyptr1_]      ; Get rid of PF7 graphics character.
        tst   busy           ; See if the following routine is still at work.
        if    eq             ; Not at work.
          dec busy           ; Set busy flag so we aren't interrupted.
          ldd #message       ; Substitute your own routine from here on.
          jsr printf_        ; Say we're in routine; ask for a keypress.
          cli                ; Let interrupts proceed so we get a keypress.
          loop               ; The BUSY FLAG will shunt any interrupts.
            jsr kyputb_      ; Get a character.
          until ne
          clr  [kyptr1_]     ; Whatever character we got, clear it!
          ldd  #proceed
          jsr  printf_       ; Tell user we're through the routine.
          clr busy           ; So clear the busy flag.
        endif
fini    rts

busy     fcb 0               ; Use all of these lines except BUSY for the next
normserv rmb 2                                                ; routine also.
discon   fcc "User IRQ Routine Disconnected%n"
         fcb 0
proceed  fcc "%nYou Have Left the Routine%n"
         fcb 0
```

```
message  fcc "%nIn User Routine%nPress Any Character Key to Proceed%n"
         fcb 0
         end
```

THE SEI APPROACH

```
ourtask jsr   [normserv]     ; Get normal IRQ service.
        sei                  ; Deny Moriarty.
        ldb   kyindx_        ; Load offset value of last key pressed.
        guess
          cmpb  #$59         ; Did we press SHIFT LEFT-ARROW?
          quif  ne
          clr   [kyptr1_]    ; Get rid of SHIFT LEFT ARROW character.
          ldd   normserv     ; Clear interrupt routine and resume
          std   intvctr_+8   ; normal IRQ service.
          ldd   #discon
          jsr   printf_      ; Tell user we disconnected special routine.
        admit
          cmpb  #$95         ; Did we press PF7 keys?
          quif  ne
          loop               ; Yes, we're still holding them down.
            jsr $DEF3        ; $DEF3 is system routine KYSCAN_.
            tstb             ; Wait until PF7 keys are released.
          until eq
          clr   [kyptr1_]    ; Get rid of PF7 graphics character.
          ldd #message
          jsr printf_        ; Tell user we're in this routine.
          loop
            jsr $DEF3
            tstb             ; Get a keypress from the keyboard.
          until ne
          loop
            jsr $DEF3        ; And wait until the key is released.
            tstb
          until eq
          clr   [kyptr1_]    ; Clear from buffer whatever character we got.
          ldd   #proceed
          jsr   printf_      ; Tell user we're out of the routine.
        endguess
        rts   ; Use all auxiliary data in BUSY FLAG except BUSY. Revise Message:

message fcc "%nIn User Routine%nPress Any Key to Proceed."
        fcb 0
        end
```

COMMENTS ON STRUCTURED CONTROL Though it's apparently simple, the FOR...NEXT or DO...ENDDO loop is deceptively complex and highly useful. It does not work in the same fashion in SuperPET's languages as it does in Microsoft BASICs. Waterloo has given us more freedom. The FOR...NEXT or DO loop is the fastest of all the loops employed in structured control. When speed is important, use it in preference to other loops which may be more readable but are much slower. In the comparisons which follow, see the large difference in times to execute when we substitute floating point or real numbers for integers, and the substantial difference between various loop structures. (Times for mBASIC and mFORTRAN by the editor, for mPASCAL by Bob Davis:)

Time to Execute:	for i=1 to 1000 v=v+1 next i	loop v=v+1 until v=1000	while v<1000 v=v+1 endloop
Real Numbers:			
microBASIC	7.53 sec.	9.52 sec.	10.93 sec.
microFORTRAN	27.5 sec.	42.4 sec.	42.6 sec.
microPASCAL	28.6 sec.	48.4 sec.	48.0 sec.
Integers:			
microBASIC	5.38 sec.	8.39 sec.	8.58 sec.
microFORTRAN	23.2 sec.	37.1 sec.	37.2 sec.
microPASCAL	21.3 sec.	35.4	35.0 sec.

Admixtures of real numbers and integers run slowest of all. The index itself (i, above) should be an integer for maximum speed. The range (1 to 1000) may be set in real numbers with no loss of speed. If in the incrementing line (v=v+1) we make v or 1 a real while the other is an integer, speed suffers. Change both to integers and execution time is cut 29%. Any admixture of real numbers and integers suffers the same disadvantage within the loop.

In all SuperPET languages, you may leave these loops at any time without resetting the index (i, above) to its final value, as is required in Commodore BASIC. The "quit" statement employed with all other loops serves here as well; there may be as many alternative exits as you require, as shown at left. Quite often, you must use flags to supplement quit statements. Suppose, for example, that we screen a mail list to pick up the last name and to send "Dear Mr. Qwerty" salutations on form letters. We certainly don't want to stay in any loop longer than needed to finish our work, so we might write the inner FOR...NEXT loop as at left, below, with an outer loop in which we get all entries from the mailing list (we don't bother with the last name for the sake of clarity). The more common titles precede the less common, and we quit the inner loop on a flag as soon as we find the right title. Puzzled by the "idx?" This function in mBASIC is matched with "index in mFORTRAN. Both find a sub-string within a larger string. If mailist$="Ms. Lira Uiop", then: idx(mailist$,"Lira") yields 5--the exact count of characters from start-of-string to the "L" in "Lira." Sometimes, as in the example at left, we don't care where a sub-string is, but only wonder if it's there at all. If it's not, both "idx" and "index" report zero, and an "if idx..." fails. A word of caution: don't use "if not idx...", for the reasons set forth on p. 116, Vol 1.

```
do i=1,40
  quitif x=<0
  quitif y>560
  ...
  ...
enddo
------------
loop
  ...
  data "Mr.","Mrs.","Ms.","Miss "
  for j=1 to 4
    read title$
    if idx(mailist$,title$)
      salutation$="Dear "+title$
      flag=1
    endif
    if flag then quit
  next j
  restore
{second example; continuation}
  if flag
    quit ! Or use flag=0 and no 'quit'.
  elseif idx(mailist$,"Dr.") or idx(mailist$,", M.D.")
    salutation$="Dear Doctor: "
  elseif idx(mailist$,"Rev.") or idx(mailist$,"Reverend)
    salutation$="Dear Reverend: "
  ... and on and on and on...
  endif
```

The example continues, to show how a "quit" from a FOR...NEXT loop carr-

```
  ...{We jump here on the "if flag quit"}
  ...
endloop
```

ies over into an IF...ENDIF. Having screened out the common Misters and Mizzes, we don't want to wade thru a long list of less common titles; the "quit" on flag in "second example" makes sure we jump to the line following "endif" if indeed we've already found a Mr., Ms., etc. Note well that the "quit" in an IF...ENDIF structure must be on a line by itself.

Many have tried (in vain!) to change the values of the range in DO or FOR...NEXT loops (in "do i=1,1000", the range is 1 to 1000). The range is set when the loop is entered and cannot thereafter be varied. Yet the value of the index (i) may be changed in microBASIC, but not in microFORTRAN, where any attempt results in an error. We might, for example, want to get rid of null lines in a disk file brought into memory. Since a null line is a blank line on screen and to printer, this is a common problem. We solve it at left, below, by simply overwriting any null line, including the null at EOF, with a line which has some string value--and we do it very easily by subtracting 1 from index i whenever we encounter a null line. If we finish the loop before the last line comes off disk, we'll find one blank, null line in the printed array line$--the very last line, for i will have the terminal value of 1001, not 1000, and line$(1001) will be null. If the terminal index value of i is later used in program, remember to reduce it by 1 if, and only if, we haven't left the loop prematurely with a "quit". This holds for mBASIC and mFORTRAN.

```
for i=1 to 1000
  linput #40, line$(i)
  if io_status <> 0 then flag=1
  if line$(i)='' then i=i-1
  if flag then quit
next i
```

We find one other way to leave these loops in microBASIC, but not in mFORTRAN--set the index i to its final value, as shown at left. If you will not later need the value of i as an index, this is a simple way to quit the loop.

Be warned that the relative speed advantage of integer FOR...NEXT and DO loops dwindles as the amount of code within the loop is increased. You are hard put to find any speed difference between an WHILE...UNTIL, LOOP...UNTIL, or other formal loop structures, and a FOR...NEXT loop if a great deal of complex code lies between the start and end of the loop, or if the program is "bound"--bound by disk input/output capacity, by operator reaction time, or by printer output capacity. On the other hand, numeric calculations executed within loops are far faster if executed as FOR...NEXT or DO loops, using integers whenever possible, than in any other form of loop structure.

```
for i=1 to 1000
  linput #40, line$(i)
  if io_status <> 0 then i=1000
  ...
next i
close #40
```

* * *

Anent FOR...NEXT loops, we got a note from Loch Rose which points out that if you leave such loops prematurely in BASIC 4.0 (ten times is the limit) you will fill the stack and get an error message. Loch continues: "I wrote a 6809 machine language routine to print out the values of the system stack pointer and user stack pointer and jumped out of 100 loops, each one using a different counting variable, SYSing to the ML routine at intervals. The answer is that FOR loops, just like all the other loops, don't use the stacks; stack pointers were completely unaffected by my antics, remaining the same in and out of loops. The manual says you can QUIT a 'repetition' structure; don't be bashful about leaving FOR loops any way you choose." We've done the same in mFORTRAN, and know you may leave DO loops at any time without ill effect.

REFERENCE STUFF Readers have asked what books will help in learning the languages in SuperPET. Steve Zeller has commented on available material for APL in his column from time to time. As readers tell us about other books in other languages, we'll report. Please send in your opinion of the books you've used--good or bad. We've added other reference material on 6809 and 6502 machine language, the serial port, HOSTCM, etc., below.

PASCAL (Recommendations by Bob Davis, Associate Editor, PASCAL):

Problem Solving and Structured Programming in PASCAL, Elliot B. Koffmann, Addison-Wesley (1981). Contains many examples and has answers to many exercises at the back. It is written in relatively "pure" PASCAL so there's a minimum of modification required for SuperPET. It includes a section which explains the UCSD extensions to standard PASCAL; the author includes suggestions on which sections to study, depending on interests of students and instructor.

Programming in PASCAL, Revised Ed., P. Grogono, Addison-Wesley (1980). I find that this book moves a bit rapidly and is a bit sophisticated for me to use for learning PASCAL. However, I frequently find it a useful reference.

Most other texts I know of are aimed specifically at UCSD PASCAL and require considerable modification and interpretation for use with SuperPET.

MicroFORTRAN

Recommended by Associate Editor Stan Brockmen: Problem-Solving & Structured Programming in FORTRAN, 2d Ed., Frank L. Friedman & Elliot Koffman, Addison-Wesley (1981). Thumb-indexed, goodly number of examples, exercises with answers in back of book; written to FORTRAN 77 standard; with exception of string handling (not done to 77 standard on SPET), most everything useful in learning mFOR.

Added by Editor: Structured Fortran 77 Programming, Seymour M. Pollack, Boyd & Fraser, San Francisco (1982). Comprehensive, oriented toward mainframes; handles structure well. Many examples; exercises with answers in back of a large, thick, well-illustrated volume.

MicroBASIC (Recommended by the Editor):

Structured BASIC and Beyond, Wayne Amsbury, Computer Science Press (1980). While it applies structured concepts to Microsoft BASIC, this book shows in some depth how to handle structure, and employs a pseudo-code to block out programs. The pseudo-code is very similar to the structured controls available in SuperPET languages. Exercises and a review follow each chapter; answers to exercises are found in the back of the book. A good introduction to structured concepts and to the essentials of the language.

MicroCOBOL (Recommendations by Steve Starwich):

"The Dirksen-Welch manual (comes with SuperPET) is the best I've seen. If you work through the tutorials, you learn COBOL. Here are some others:"

Structured COBOL, Grauer, Prentice-Hall, Englewood Cliffs, New Jersey, (1981).

A Programmer's Guide to COBOL, and A Guide to Structured COBOL, Lim, Reinhold Co., New York N.Y. (1980).

COBOL Support Packages, Naftaly et al, Wiley, New York, N.Y. (1972).

COBOL Programming, Watters, Heinemann Educational, London (1970).

Other Recommendations from Various Sources, Including Dyadic Resources Corp.

Programming the PET/CBM, Raeto West, COMPUTE! Books, Greensboro, N.C. (198?). Recommended by Terry Peterson, particularly for ROM and DOS routines in SPET.

Waterloo BASIC Primer and Reference Manual, McPhee, Graham, Welch, WATCOM Publications, 415 Phillip St., Waterloo, Ontario, Canada N2L 3X2 (1980).

COBOL Programming: A Structured Approach, Abel, Reston Publishing Company, Reston, Virginia (1980).

WATFIV: Fortran Programming with the WATFIV Compiler, Moore, Reston Publishing Company, Reston, Virginia (1975).

The Commodore SuperPET Computer - Machine and Assembly language for the Motorola 6809, Cowan (not dated), WATCOM Publications (address above).

MCS6500 Microcomputer Family Programming Manual, 1976, MOS Technology, Inc., 950 Rittenhouse Road, Norristown, PA 19401 (6502 Machine Language).

HOSTCM Specification Document, Wilkinson, WATCOM Products (address above).

The SuperPET Serial Port, J. Scheuler, WATCOM Products (address above), 1984, $20 postpaid, Canadian in Canada, U.S. elsewhere. Excellent reference.

WATCOM News, 415 Phillip Street, Waterloo, Ontario, Canada N2L 3X2. Monthly, Subscription $10 Can. in Canada, U.S. elsewhere. (Replaces infoWAT and WATNEWS).

Keysoft International Limited offers education services and software for SuperPETS. 8111 Yonge St., Unit 4, Thornhill, Ontario, Canada L3T 4V9.

A BUG IN microFORTRAN INPUT

by Stan Brockman, Associate Editor
11715 West 33rd Place
Wheat Ridge, Colorado 80033

I may have given too little attention to a problem described by P.J. Rovero in I, 93, in which the last field of an input record does not come off disk properly; although the solutions given there and in my article in I, 183 indeed work, Rovero's problem is symptomatic of a bug in m-FORTRAN which this article attempts to define. SuperPET is very persnickety when the actual length of an input string is less than the expected length. The short program at left demonstrates what I mean. If you enter any less than five characters, nothing happens until you hit RETURN a second time. If the second entry is null or also too short, you receive a "field too wide" error. Make an entry of five or more characters, though, and execution continues (in this case, the first entry is either dropped or ignored). I had recently begun to think that this was a bug in the input conver-

```
character a, fmt1
real b
integer i

print*,"Enter some Characters"
fmt1="(a5)"
read fmt1,a
print*,a
end
```

sion process for character variables, but then changed my mind. Note that if the first or second try is too long, it is truncated on the right, consistent with standard FORTRAN. If you add your own blank-fill to the right of a short entry, it doesn't correct the error unless you put a non-blank character somewhere out to the right. Does this begin to sound like P.J.'s problem? It is indeed the same; I recently discovered that the handling of short inputs causes problems in all types of input data, not just with character data, and from both disk and keyboard input. (You can recover from an incorrect keyboard entry. Simply tell the debugger to "cont", cursor back up to your entry, change it as necessary, and hit <RETURN>. You're back in business!)

If you revise the program above to use "i5" format in place of "a5", and read an integer variable with five or fewer digits, you'll get the same error. Use fewer than five digits, right-fill with blanks, end with a non-blank character, and you get a number with trailing blanks converted to zeroes--consistent with the way FORTRAN is supposed to work. See the example at the left, where a four-digit entry followed by a blank and a comma is accepted and LD prints "12340" as the value input (^ stands for a blank or space). If you enter numbers right-justified in a five-digit field, everything is still OK.

```
*Accepted:
1234^,
*Fails:
1234^
```

Change the program again, this time to use real data (decimal numbers) with format of "f5.2" and read some reals with and without a decimal point. You'll get the same error with a short input. With a decimal point, any number of digits in front of or behind the decimal will be okay so long as it is right-justified in its field or a non-blank is hung out to the right somewhere. Without a decimal point, the integer discussion above applies (except that an implied decimal will appear in its proper position, to give the number two decimal places).

The examples above read one-field records. Multi-field input records receive the same shabby treatment from mFOR. If the last field is too short, the bug bites you. The errors discussed so far apply to disk input as well, but are evidenced differently (see below). The fixes on pp. 93 and 183 of the Gazette apply to both keyboard and disk input.

In short, mFORTRAN appears not to be watching for end-of-record marks. The input algorithm demands that the input count be equal to or greater than the width of the field, but obviously does not blank-fill to the right if it encounters the end-of-record before the count is correct. The versions of FORTRAN with which I am more familiar instead accept short inputs, teminating whatever is entered before the EOR, at which time a character variable is blank-filled and both real and integer data are translated as though the last digit were the least significant digit. mFORTRAN, it appears, is not smart enough to terminate a record as it is input, as does standard FORTRAN.

The same problem appears on input from disk of a series of values (See Rovero, I, 93). While reading a disk input record, mFOR matches data to the corresponding variables. If there are more values than variables (not the problem in Rovero's bug), the excess of data is discarded. Conversely (and true of Rovero's example), if there are not enough values, mFOR reads another record, if it can, in order to satisfy the variables which don't yet have value. In the example at left, above, the records are two

```
*Index to bytes:
123456789012345678901234567890
    5.6       6.7       8.8
    4.2       4.8       9.2
```

characters short when read with "read(1,'3f10.1') a,b,c". The variables a and b receive their correct values, but the third field [8.8], being short, is discarded by the bug; mFOR proceeds to read the next record [4.2] to satisfy variable c, and the balance of the record is discarded. A second read then starts with the third record, and so on. You can see from this example that the bug indeed makes hash of your input data!

While I'm not sure that all of the above really describes a bug, it certainly will do until one comes along. mFORTRAN simply does not know how to deal with a short input. It ought to generate an error message immediately instead of seemingly dying. Alternatively, it should be generalized enough to handle this sort of problem. The requirement that the last field in a formatted read may not be short is not documented so far as I know. The user is left with a problem when variable-length text files (the default type, recall) are used as input to mFORTRAN. I trust Waterloo will eventually put an end to what I call Waterloo's Waterloo--with a patch or revision to the language.

ANATOMY of MICROBASIC
Part 3

by Gary Ratliff, Sr.
215 Pemberton Drive
Pearl, Mississippi 39108

Before we jump into the disposition of DIMensioned variables, let us review the tokens and conventions we have so far uncovered. Waterloo uses codes above decimal 128 as tokens. $95, for example, is "=" or, more properly, "is assigned the value of." $96 means "multiply by". $8D stands for "numeric" type of variable; whether integer or real (floating point). $84 is a "string" variable; $D1 identifies a "function." We find some new tokens this time: $D6 tells SuperPET to DIMension a variable, whether integer or real. $9C identifies a variable as a DIMensioned one. Last, $9D tells SPET that the value is found in the table of DIMensioned variables.

We may have solved another mystery. Remember that the end of mBASIC lines always is identified by 04 ff ff 03 immediately after the last line of the program. I thought the 04 03 was a conventional forward/backward pointer, counting bytes in the entry, until I discovered that same form (04 xx ... xx 03) marks the beginning and the end of the huge table where DIMensioned variables are stored! Note issue 14, p. 251, where we see a "pointer/separator, not defined." Aha! We have now defined it. Yes, the values or addresses of DIM variables are contained in a separate table, not in the Single Variable Table we examined last issue. The DIM variable table starts immediately after the 04 ff ff 03 which marks the end of mBASIC program lines.

To make all this clear, let us enter and run the simple program at the left. We see below the program as dissected; then the DIM variable table; and, finally, the entries in the Single Variable Table, which define the names of the DIMensioned variables, though the values are in a separate table of DIMensioned variables. Please note that the last DIMensioned variable appears first in the table; variable b%, although last in our program, appears first in the table.

```
10 dim a%(10)
20 dim b%(10)
30 a%(0)=2
40 b%(0)=4
50 b%(1)=a%(0)*b%(0)
```

I suggest you run through the dissected program below and then come back here to read these comments, which will not otherwise make much sense!

As with other variables, the first DIMensioned variable is numbered $01 for the location of its name in the Short Variable Table; the name of the second (b%) is found 5 bytes into the Short Variable Table, and so it is Variable 05. The valu-

es or addresses of variables are often found in the Short Variable Table, but we don't find them here. DIMensioned variables can have so many values that there's no room in the Short Table. The problem for Waterloo (and for us) is then how to cross-reference the names, the locations and the values of DIMensioned variables. It turns out the solution is relatively straightforward:

The names of DIM variables are found in the Short Table, with nulls for values. The last dimensioned variable being found first in the DIM table, its location in the DIM table is cross-referenced in the Short Table. Well, if we can find the last DIM variable, we can easily find the rest--because the DIMension value of each variable is found in the DIM table! If, for example, we know that b% has a dimension of ten [plus one for b%(0)!], we simply count forward the eleven spaces or blanks assigned for values, and we're at the next variable. (Integers need 22 bytes for 11 values; reals need 5 x 11, or 55 bytes). Whichever, it's easy to count forward or backward to find the DIM variable we need. And it is equally simple to find the specific variable [say b%(4)]--we simply offset from the start of the DIM variable by 4!

	LoL	Start of mBASIC		Bptr	LoL	Line	10	DIMension
;0a01	04	00	00	03	0b	00	0a	d6
	Var	at 01	Type DIM	Numeric	10	in Table	Bptr	LoL
;0a09	00	01	9c	8d	0a	9d	0a	0b
	Line	20	DIM	Var	at 05	Type DIM	Numeric	10
;0a11	00	14	d6	00	05	9c	8d	0a
	in Table	Bptr	LoL	Line	30	Var at	01	Type DIM
;0a19	9d	0a	0d	00	1e	00	01	9c
	Numeric	(0)	in Table	=	Numeric	2	Bptr	LoL
;0a21	8d	00	9d	95	8d	02	0c	0d
	Line	40	Var	at 05	Type DIM	Numeric	(0)	in Table
;0a29	00	28	00	05	9c	8d	00	9d
	=	Numeric	4	Bptr	LoL	Line	50	Var
;0a31	95	8d	04	0c	18	00	32	00
	at 05	Type DIM	Numeric	(1)	in Table	=	Var at	01
;0a39	05	9c	8d	01	9d	95	00	01
	Type DIM	Numeric	(0)	in Table	mul. by	Var at	05	Type DIM
;0a41	9c	8d	00	9d	96	00	05	9c
	Numeric	(0)	in Table	Bptr	End of mBasic Lines			
;0a49	8d	00	9d	17	04	ff	ff	03
	Start Pointer to DIM table		DIMens'd	Start of DIMensioned Variable Table to 10	Store b%(0)		Store b%(1)	
;0a51	04	00	00	0a	00	04	00	08

```
;0a59 through 0a6a:     Storage for remainder of ELEVEN b% values
                  b%(2)          b%(3)          b%(4)          b%(5)
;0a59       00       00    00       00    00       00    00       00
                  b%(6)          b%(7)          b%(8)          b%(9)
;0a61       00       00    00       00    00       00    00       00
-------------------------------------------------------------------------------
                  b%(10)    |    DIM to 10       Store a%(0)     Store a%(1) etc.
;0a69       00       00    |  00       0a    00       02    00       00
-------------------------------------------------------------------------------
       A number of reserved lines for DIM variable a% are deleted here.
-------------------------------------------------------------------------------
The 03 marks the  |            Single Variable Table: 0a84-0a94
end of DIM table  |Int. Var   Name                  Integer Var  Name
                  |of len=1     a                   of len=1       b
;0a84       03    |  41       61    00    00       41       62    00
-------------------------------------------------------------------------------
Because last DIM variable b% is first in the DIM table, its location is defined:
                 4 Bytes  from start of table with space for a total 11 values
;0a8c       00       04    00       01 (?) 00       00       00       0b
-------------------------------------------------------------------------------
                  |    Cross-Reference Table. References NAMES and LOCATIONS
                  |  Find   Variable 01 at 00   Variable 05 at 5
 End Var. Table   | (token) bytes, Table Above  bytes, Table Above.
;0a94       ff    |  e7       00       01    00       05       00       00
```

[Ed. Loch Rose has used Gary's dissections of microBASIC to prepare a machine-language routine named FINDVAR. It locates strings in microBASIC and will transfer them to and from assembly language programs. We'll say more next issue.]

CURRENT and AVAILABLE SPET DISKS Since we've had a number of requests for a current list of ISPUG disks, here it is, with references to the Gazette pages which define what's on disk. Most disks may be obtained from the Editor, at PO Box 411, Hatteras, N.C. 27943, in either 4040 or 8050 format. We include on the list commercial software of use with SuperPET. If you order, please state 4040 or 8050 format!

PAPERCLIP. Commercial. $150 ($60 discounted); excellent WP program, see I, 173, 238, 244. Available from Batteries Included (authors), 186 Queen Street W., Toronto, Ontario, Canada M5V 1Z1, and from A.B. Computers, 252 Bethlehem Pike, Colmar, PA 18915 (last reported available for $60). Version 9000A.

ISPUG MASTER TELECOM. Three versions: 1) Complete, with both 6809 and 6502 programs, defined in I, 135 ff, 185: $25. Has done yeoman duty for many. 6809 version only, $15; 6502 version only, $15. Full instructions, full data on wiring RS-232 port/modem. From Editor.

COM-MASTER. Commercial. $95. Defined in I, 136, 210, 222 ff. Excellent general purpose telecom program, interrupt-driven; also works splendidly with APL. From Quality Data Services, 2847 Waialae Ave., Honolulu, Hawaii 96826.

PETCOM. Commercial. $99 Can. Reviewed in I, 229. Excellent general purpose telecom program, not suited for APL. APL Version available. Inquire of author at Ph.D. Associates, Suite 200, Kinsman Bldg., Downsview, Ontario, Canada M3J 1P3.

SUPERPET BULLETIN BOARD. Written by ISPUGger Paul Matzke in microBASIC to enable bulletin-board ops in 6809; makes you the SYSOP of a system which will handle TC with all other computers. 4040 format from Paul V. Matzke, PO Box 574, Madison, WI 53701, or in 8050 from Editor. $6 U.S. Noted in I, 161.

HOME ACCOUNTING. Available only in 8050 format (won't fit 4040 disks) from the author, Delton B. Richardson, 4299 Old Bridge Lane, Norcross GA 30092. An

excellent way to handle personal finances; with tutorial and instructions plus bar-graph system. Defined in I, 146, 213. $15 U.S.

MICROPIP. Commercial. Best program yet to handle files. Reviewed in I, 129. If you have to sort, copy, compare or evaluate disk files, get it from WATCOM Products, 415 Phillip St., Waterloo, Ontario, Canada N2L 3X2. $75 Can. in Canada and $75 U.S. elsewhere. Superb when you compare and locate differences in ML.

APL CHARACTER SET. Workspaces, character sets for printing APL to EPSON and to Commodore 8023 printer, with instructions. Defined in I, 196, 207. Plus some APL utility programs. $10, from Editor.

STARTER-PAK. For the user new to SuperPET, not an introduction to computers. Defined in I, 187; disk and manual; many utilities; $15 U.S. From Editor only. Input/output programs all languages but COBOL; reference sheets attached define all DOS commands, microEDITOR search/replace; disk I/O, printer output. We wish we'd had it when we opened the box.

ISPUG UTILITY. Defined, I, 213, 235. Disk crammed with useful utility programs from all over: dumps, sorts, ROM images for maintenance check; two extended monitors; bad disk retrieves, a fast microEDITOR version, disk drive address changer, etc. $10 U.S. from Editor in 8050; $16 for two disks in 4040.

APL EDA. Defined I, 89, 110. Implements John Tukey's Exploratory Data Analysis; for journeymen APLers. $10 U.S.; in 8050, from Steve Zeller, 6425 31st St., N.W. Washington, D.C. 20015; in 4040 from Editor.

APL ANSCOMBE. For those who use statistics; implements John Tukey's book on computing statistics in APL. Defined in I, 226. $10 U.S. from Editor.

HOSTCM. Waterloo program to interface SPET to mainframes or minis. Reviewed in I, 102. Available from WATCOM Products (see microPIP, above, for address).

TC FROM LANGUAGE. Defined in I,246, 262. BASICOM and APLCOM are assembly-language modules which allow the user to telecommunicate from the languages in 6809 mode at 1200 baud. You don't need to know assembly language, but must write your own program in language. With instructions and examples. $10 U.S.from Editor. UNICOM, general purpose program, on disk for mPASCAL and mCOBOL.

MicroEDITOR Disassembly/Reassembly. Defined I,181. For journeyman assembly language programmer, a disassembly and reassembly of mED which runs, by John Toebes. $6 U.S. from Editor.

OP SYS DIS. Partially commented disassembly of SPET operating system/system routines, by John Toebes and others. $6 U.S. from Editor. See I, 181.

TPUG PROCEEDS WITH OS-9 We hear that the Toronto Pet Users Group (TPUG) has decided to make the OS-9 operating system available for SuperPET, as discussed in issue 14, p. 261. OS-9 was developed by MicroWare Corporation in conjunction with Motorola as a way to take advantage of the 6809 instruction set and addressing modes.

In OS-9, you can allocate various sections of memory to separate programs. Each section has its own, individual stack. You may thus switch from program to program (and back again) without reloading programs from disk and without interference between programs. Assembly language programs are written in PIC (position independent code) so that if loaded they'll work wherever located. The 6809 was designed to provide exactly this capability.

This means that with OS-9 you can run several programs at the same time. The buzzword is concurrency. TPUG says we get a multiuser capability too, if that is important to you. Schools might find it most useful.

TPUG says that access to the Waterloo languages and programs will be preserved, and that a prototype of the OS will be operating this month (September, 1984).

Avygdor Moise of Toronto is doing most of the programming; MicroWare, the distributor of OS-9, will provide much of the documentation and support. TPUG says that there will be source code compatibility to versions of OS-9 that are planned for the Motorola 68000, and that OS-9 will give users direct access to hardware drivers which could operate parallel printers, additional serial ports, and devices such as hard drives. OS-9 is similar to, but simpler than, UNIX.

Apparently the final cost to TPUG members hasn't been quite nailed down, but is now estimated at $150 U.S. per purchaser (lower than the $200 per purchaser we quoted in issue 14). From preliminary information, we learn that some hardware changes are required. TPUG says these changes will not affect normal operations with the Waterloo languages and facilities.

The price we quote above is to TPUG members or associate members. We've seen nothing to indicate OS-9 is available to non-members. U.S. associate memberships in TPUG are $20 per year. To join, write TPUG at the address in the second paragraph following.

Apparently there'll be two different add-on boards, one for the old three-board SuperPETs, and another for the newer, two-board models. The quoted price includes both the hardware and the OS-9 software.

Let's face it: TPUG isn't going to go ahead until receives enough money to ensure that it can meet its obligations to MicroWare for a license to use the new operating system. If you want to get into the act, send $68.09 U.S. to TPUG, at 1912A Avenue Road, Suite 1, Toronto, Ontario, Canada M5M 4A1 with a letter specifying what it's for. This will reserve a copy of the OS for you, and give TPUG assurance that there's enough interest to proceed. The remainder of the cost of OS-9 will be paid when (1) the OS is available, and (2) the final cost is determined. The final cost per copy will, of course, depend on how many of you people are interested enough to order. We've ordered, if only to get our hands on 'C'.

What happens if TPUG does not proceed or there's not enough interest? TPUG says that deposits will be returned. We can't make any promises for TPUG, but the last time we looked, there were about 25,000 members and the outfit was highly solvent. Since it's a non-profit users group, this isn't a scam. We have therefore shipped off our $68.09 (get it now?) to TPUG to reserve our copy, but not without a qualm or so on how widely useful it'll be. More later on the qualms.

What do you get for your money? Not having seen or run the OS, we don't know, but it seems to have a good reputation. Since OS-9 runs on the Radio Shack Color Computer, there's quite a bit of software available, some of which we list below. TPUG says all OS-9 software will run on the SuperPET OS-9 system, and that any OS-9 program written on SuperPET will run on any other system using OS-9.

Some OS-9 Commercial Software: Two assemblers, BASIC09 (a highly structured BASIC), two PASCALS (one lean, one full), CIS COBOL, 'C' with assembler, linker, and library, STYLOGRAPH, a word processor with spelling checker and form letter capability, an accounting package, a spreadsheet, two text editors. For more details, see MICRO for December, 1983.

Apparently this is what you get with OS-9 if you purchase: An assembler, an editor, command (shell) library monitor, a symbolic debugger, hardware, and the OS itself. The languages or other programs we've listed above you'll have to obtain

separately, either as public domain or as commercial software.

One problem with software: how do you convert it to Commodore disk format? TPUG says that "it will participate in the acquisition of public domain software and assist users in the conversion of commercial software so it will operate on Commodore drives." We'd like to know more about exactly how TPUG will assist in the "conversion of commercial software," because we don't give two hoots for ninety-five percent of the undocumented public-domain software we've seen. If you're in a neighborhood user group, you often can find somebody who can explain undocumented stuff. If you're not, in our experience, it takes less time to write your own program than to decipher the unexplained material on public domain disks. We intend to explore the matter of converting commercial software in much more detail with our members in TPUG. We can only hope the public domain data is better documented than the usual run of such stuff; ISPUG members are scattered from the Canadian Barrens to the Indian Ocean and cannot get support by a local phone call. We trust TPUG will remember this and document OS-9 well.

In the end, our qualms center around support for OS-9. If you're an assembly language expert, the OS should present no large problems, but if not, you're going to need support. The Gazette has provided that support for the Waterloo languages and facilities, but ye ed has neither the time nor money to support OS-9, nor is he able to clone and publish a Gazette for OS-9 users. Let's face it: those who don't have OS-9 aren't going to read about it; those who do will need a journal. Which of you is going to write it? Or is TPUG? TPUG charged ahead without resolving this issue, which we frankly think will be ultimately critical to the success of OS-9 outside of Toronto and its suburbs. Are there any volunteers who'll publish an OS-9 supplement to the Gazette?. If you'll write it, we'll be glad to publish and distribute it to special subscribers.

In sum, we're going with OS-9 because for $150 or thereabouts we'll get an advanced capability for SuperPET without shelling out for a new computer, and because we'll open the door to commercial software not otherwise avaiable on our machine. Congratulations to TPUG for the courage and vision to proceed.

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	

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)

Name:_____ Disk Drive: _____ Printer:_____

Address:_____

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

[] Check if you're renewing; clip and mail this form with address label, please.

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.

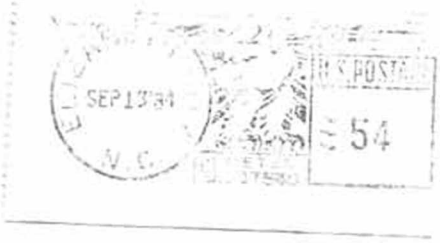
SCHOOLS: Send check with Purchase Order. We do not voucher or send bills.

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.
 If you send inquiries, enclose a self-addressed, postpaid envelope (4 x 9.5 inches, please). If you submit material for the Gazette, enclose a suitable return/reply envelope, postpaid. Canadians: enclose Canadian dimes for postage. See enclosed application form for membership dues. The Gazette comes with membership.

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.

FIRST CLASS MAIL

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



First-Class Mail
in U.S. and Canada