JANUARY 1986
VOL.7, NO. 1

PEEK (65)

The Unofficial OSI Users Journal

P.O. Box 347
Owings Mills, Md. 21117
(301) 363-3268

INSIDE

Column One

Happy New Year! Along with our best wishes for 1986 comes both exciting news and sadness too.

Do you remember 1981? That was the year Karin had PEEK dumped in her lap and was told to get it into the mail in two weeks! Well, that might not be so bad for someone well schooled in computers or publishing. Frankly, it was asking a bit much. Nonetheless, do it she did! Shortly, thereafter, PEEK grew to its present size and experienced numerous improvements along the way.

That was five years ago. Five years of monthly deadlines that have become more demanding only because of the complexities and other involvements of our lives that have caused us to carefully, selectively, and quietly look for a new leader.

The time has now come when all the pieces fit together in perfect harmony to make the transition to new leadership and ownership.

The very thought of giving up PEEK is a little like sending your first child off to college. It leaves an empty place. We certainly will miss the personal, encouraging calls and letters that have spurred us on during these years and we hope that we will continue to hear from you from time to time.

During the past couple of years, if there was one person who stood out in the continuing support of PEEK, it was Rick Trethewey. Not only has he supplied our readers with a veritable wealth of information in his acclaimed articles, but he has also been a valuable technical assistant upon whom I have leaned many times.

What better person could there possibly be to take over the reins? A past employee of OSI, the Systems Operator of the CompuServe's OSI Special Interest Group, an innovator of software, master of OS-D, and fast becoming one of OS-U. These are the things that make us very comfortable in knowing that PEEK will continue in excellent hands.

We expect to be working very closely with Rick for the next few months to make the transition smooth and bug free and hope that you will give him the same kind of support that you have given us. There is a definite need for the information contained within these 24 pages. PEEK <u>MUST</u> continue to flourish, and will do so with the continued support from each of you. Make it a New Years Resolution to contribute to YOUR magazine. Spread the word about PEEK, - make your friend buy his own subscription, share your knowledge and allow others to gain from it such as you have. Dealers, make the time to share your thoughts and problems, - you have nothing to lose, only to gain from each other. Also, advertise your wares, where else do you have such a captive audience?

Rick has not told us about all of his new ideas, but those that we know about sound like good solid improvements that will make PEEK better than ever. There will certainly be changes, --- but maybe it is time for some new ideas!

You may want to write down the new address and phone number for PEEK(65), effective as of January 1, 1986.

PEEK(65)
P. O. BOX 586
PACIFICA, CA 94044
(415) 993-6029

In the meatime, anything that does find its way to our door will be expeditiously taken care of and/or forwarded to Rick. Our phone will continue to be operative for a while, and we will be happy to answer any questions you might have. As I said before, this will be a smooth transition.

Rick, we wish you every success and hope that you will have as much pleasure being so closely involved with the OSI/DBI world as we have.

Last, but not least, my gratitude to all our subscriber, advertiser and author friends, plus the past and present PEEK staff, without whose support and encouragement PEEK would not be what it is today.

Eddie

DIRECT BOOT

By: D. G. Johansen
P. O. Box 252
La Honda, CA 94020

Listing 1 modifies OS65D to allow you to directly boot to assembly code of your choice. By using this modification, you may run your code by inserting a disk, containing assembly code on track 02 (normally containing BASIC), pressing BREAK, followed by "D" to LOAD and GO to object code instead of BASIC.

This article discusses some of the pitfalls of booting your own code. You should not have any problems if your system operates under OS65D v3.1.

Prior to developing this routine, I had spent many hours trying to unravel OS65D. My motivation was to directly boot BETA/65, instead of using BEXEC* which requires several steps, including insertion of two disks plus several keystrokes. I finally solved the problem by purchasing the OS65D v3.2 DISASSEMBLY MANUAL by Software Consultants. This manual provides clear documentation of OS65D and has more than paid for itself in saved time. I was initially worried that v3.2 would differ significantly from v3.1 and that use of 5" disk (instead of 8") would make this manual difficult to use. These differences turned out to be minor, and I can recommend this product to any OSI user wanting to get more from OS65D, whatever version, disk size, or terminal setup your system uses.

Listing 2 contains a short program to demonstrate the direct boot routine. It is assumed that OS65D v3.1, including disk copier, is available to implement the following procedure:

Copyright • 1986 PEEK (65) Inc. All Rights Reserved.
published monthly
Editor - Eddie Gieske
Technical Editor - Brian Harston
Circulation & Advertising Mgr. - Karin Q. Gieske
Production Dept. - A. Fusselbaugh, Ginny Mays

Subscription Rates	Air	Surface
US		$19
Canada & Mexico (1st class)		$26
So. & Cen. America	$38	$30
Europe	$38	$30
Other Foreign	$43	$30

All subscriptions are for 1 year and are payable in advance in US Dollars.

For back issues, subscriptions, change of address or other information, write to:

PEEK (65)
P.O. Box 347
Owings Mills, MD 21117 (301) 363-3268

LISTING 1

```
10                 ; ****************************
20                 ; * DIRECT BOOT ROUTINE ******
30                 ; * -SOURCE SAVED ON TRACK 38*
40                 ; * -OBJECT MODIFIES TRACK 00*
50                 ; * MODIFIES OS65D TO PERMIT *
60                 ; * LOAD AND GO OF MACHINE ***
70                 ; * CODE SAVED ON TRACK 02 ***
80                 ; * BY USING ( BREAK-D ) *****
90                 ; ****************************
100                ; ****************************
110                ;
120 00E1=          OSIBAD=$E1
130                ;
140 2321=          INDST =$2321
150 2322=          OUTDST=$2322
160                ;
170 2A51=          OS65D3=$2A51
180 2AC6=          DEFDEV=$2AC6
190                ;
200 2CE4=          BUFBYT=$2CE4
210 2E1E=          OSBUF =$2E1E
220                ;
230 2C22=          XQT   =$2C22
240                ;
250 4294                 *=$2294+$2000
260                ;
270 4294 A21E             LDX #OSBUF
280 4296 86E1             STX OSIBAD
290 4298 A22E             LDX #OSBUF/256
300 429A 86E2             STX OSIBAD+1
310                ;
320 429C AEC62A           LDX DEFDEV     ; LOAD DEFAULT DEVICE
330 429F 8E2123           STX INDST      ; SET DEFAULT INPUT
340 42A2 8E2223           STX OUTDST     ; SET DEFAULT OUTPUT
350 42A5 D00C             BNE MSSG
360 42A7 F00A             BEQ MSSG
370                ;
380 42B3                 *=$22B3+$2000
390 42C4           MSSG  *=$22C4+$2000   ; RUN MSSG CODE IN OS65D
400                ;
410 42C4 A000      SETBUF LDY #0
420 42C6 8CE52C           STY BUFBYT+1   ; RESET BUFFER INDEX
430                ;
440 42C9 A930             LDA #'0        ; PUT '02' IN BUFFER
450 42CB 8D1E2E           STA OSBUF
460 42CE A932             LDA #'2
470 42D0 8D1F2E           STA OSBUF+1
480 42D3 A90D             LDA #13
490 42D5 8D202E           STA OSBUF+2
500                ;
510 42D8 20222C           JSR XQT        ; EXECUTE TRACK 02
520 42DB 4C512A           JMP OS65D3     ; START OS65D
530                ;
540                ;
550                ; *********************************************
560                ; * NOTICE-USE THE FOLLOWING PROCEDURE  ******
570                ; *      TO MODIFY TRACK 00 FOR DIRECT BOOT **
580                ; * (1)          (BOOT TO OS65D V.3.1)   ****
590                ; *                                          *
600                ; * (2) *CA 0200=13,1    (LOAD COPY ROUTINE) **
610                ; * (3) *GO 0200         (GOTO COPY ROUTINE) **
620                ; *                                          *
630                ; * (4) ?R4200           (READ TR 00 TO 4200) *
640                ; * (5) *EXIT            (EXIT TO OS65D)      *
650                ; * (6) *ASM             (LOAD ASM)           *
660                ; *                                          *
670                ; * (7) .!LO 38          (LOAD ABOVE SOURCE)  *
680                ; * (8) .A3              (ASM ABOVE CODE)     *
690                ; * (9) .EXIT            (EXIT TO OS65D)      *
700                ; *                                          *
710                ; * (10) *CA 0200=13,1   (RELOAD COPY ROUTINE)*
720                ; * (11) *GO 0200        (GOTO COPY ROUTINE)  *
730                ; * (12) ?W4200/2200,8   (WRITE TR 00 MODS)   *
740                ; *********************************************
```

Listing #2 on page 3.

LISTING 2

```
10                     ; *****************************
20                     ; * TEST PROGRAM DEMONSTRATING *
30                     ; * USE OF DIRECT BOOT ROUTINE *
40                     ; * -SOURCE SAVED ON TRACK 39 **
50                     ; * -OBJECT SAVED ON TRACK 02 **
60                     ; *****************************
70                     ;
80 3279=               HEADER=$3279
90 D41A=               SCREEN=$D41A
100                    ;
110 3279                    *=HEADER
120                    ;
130 3279 7E32               .WORD START
140 327B 9232               .WORD END
150 327D 01                 .BYTE END-START/2096+1
160                    ;
170 327E 4C8132        START  JMP BEGIN
180 3281 A004          BEGIN  LDY #4
190 3283 B98D32        LOOP   LDA HELLO,Y
200 3286 991AD4               STA SCREEN,Y
210 3289 88                   DEY
220 328A 10F7                 BPL LOOP
230                    ;
240 328C 60                   RTS
250                    ;
260 328D 48            HELLO  .BYTE 'HELLO'
260 328E 45
260 328F 4C
260 3290 4C
260 3291 4F
270                    END
280                    ;
290                    ;
300                    ; ***********************************
310                    ; * REMINDER-SOURCE CODE ALSO STARTS *
320                    ; * AT $3279. TO PROPERLY ASSEMBLE  **
330                    ; * THIS CODE FROM ASM-        *******
340                    ; * (1) .M1000  (SETS MEM OFFSET)  ***
350                    ; * (2) .A3     (ASSEMBLES TO $4279) *
360                    ; * (3) .EXIT   (EXIT TO OS65D )     *
370                    ; * (4) *RE EM  (RESTART EM )        *
380                    ; * (5) :M3279=4279,4A79  (MOVES  ****
390                    ; *      OBJECT TO SOURCE SPACE)  ****
400                    ; * (6) :EXIT    (EXIT TO OS65D)  ****
410                    ; * (7) *PU 02   (SAVES OBJECT)  *****
420                    ; * (8) *ASM     (RELOAD ASM)  *******
430                    ; ***********************************
```

1. Copy tracks 00 and 01 of v3.1 onto a blank disk, using the diskette copier.

2. Modify track 00, using the instructions given at the end of Listing 1.

3. Save the test program on track 02, using the instructions given at the end of Listing 2.

To test: Insert the prepared disk into your drive and press BREAK, followed by "D". This should cause the screen to clear and the letters "HELLO" to appear in mid-screen.

NOTES ON USE

The diskette copier requires two drives to copy tracks above track 00. If you have only one drive, you may copy both tracks using a v3.3 tutorial disk. This disk contains a single disk copy routine.

You must load track 00 code out of normal OS65D space to maintain proper function of the resident system. Follow directions carefully.

To avoid destroying source code while assembling Listing 2, you must assemble outside of source space and move the object to the load address. Again, follow directions carefully.

You might ask why line 170 jumps to the next instruction apparently wasting three bytes of code space. This is good practice because normally a boot routine must work for several different models, and custom code segments used for each model may be accessed by changing only the two bytes in the JMP address.

This boot routine has not been tested with v3.2 or v3.3. Note that v3.2 must go directly to SETBUF instead of MSSG. Also, v3.3 must boot to loaded program at $3A7E instead of $327E. The Jul. '83 and Jan. '84 issues of PEEK(65) explain OSI ROM routines used for C1P and C4P (also C8P) respectively.

★

ANOTHER SCREEN DISSOLVE UTILITY FOR THE C1P

By: Herbert H. Grassel
12838 Flack Street
Wheaton, MD 20906

This program was written in an effort to fill in some of the void between the super fast and super slow screen clears available for the C1P. Although primarily designed for use with a GRAFIX SEB-1 High Resolution Graphics board, the program can be adapted to erase any OSI memory mapped screen simply by changing the data stored at the locations listed in Table I.

On ROM based systems, the program fits into the unused RAM from $0222 (546) to $02FA (762), just below the BASIC workspace, ($0235 to $02FA for CEGMON ROM's). The program is called using the USR(X) function. Initial entry is at $0270 (624). For Disk based systems the program must be relocated. On a 32K system, for example, changing the Top of Memory from $8000 (32768) to $7F00 (32512) partitions off 256 bytes. This simplifies modifying the machine code, because only the high bytes of the internal jumps need to be changed.

The rate of dissolve is controlled by a delay loop at the beginning of the program and can be adjusted by loading a value between 1 and 255 into Dissolve Rate. This corresponds to an erase time from instantaneous to 2.5 seconds.

To accommodate the many SEB-1 display modes requires changing only two bytes; the high byte (HB) of the End of Screen Memory address and the dissolve Character Code. For a Mittendorf HRG board the HB of the beginning of screen memory must also be specified.

To dissolve the standard 24 character OSI screen, the Beginning (HB) and End Screen Memory (HB) must be changed to $D0 (208) and $D4 (212) respectively. For a blank screen the Character Code should be changed to $21 (32). With a 48 character display, the line length must also be changed to $40 (64).

TABLE I

Parameter	Location	
Dissolve Rate	$0244	580
Begin Scrn Mem (HB)	$0249	585
Character Code	$024E	590
End Scrn Mem (HB)	$0250	592
Line Length	$0258	600
Call Address	$0270	624

The following BASIC program partitions off the top 256 bytes of memory, loads the program, prints the new addresses for the parameters listed in TABLE 1, sets the USR(X) vectors - then clears itself. Line 5 executes the program to clear the screen.

```
1 RESTORE:A=PEEK(134)-1:POKE
  134,A:B=A*256
2 POKE11,112:POKE12,A: REM
  SET UP USR(X)
3 FORC=64TO120:READD:IFD=2
  THEND=A
4 POKEC+B,D:NEXTC
5 A=PEEK(B+68):POKEB+68,1:Y=
  USR(X):POKEB+68,A
6 PRINT"DISSOLVE RATE";TAB(18)
  ;B+68
7 PRINT"BEGIN SCRN MEM HB";TAB
  (18);B+73
8 PRINT"CHARACTER CODE";TAB(18
  );B+78
9 PRINT"END SCRN MEM HB";TAB(1
  8);B+80
10 PRINT"LINE LENGTH";TAB(18)
   ;B+88
11 PRINT"CALL ADDRESS";TAB(18)
   ;B+112
20 NEW
50 DATA202,208,253,162,50,136,
   208,248,169,128,141,85,2,
   169,0,160,152
60 DATA162,0,157,0,128,232,224
   ,32,208,248,238,85,2,204,85
   ,2,208,238
70 DATA173,84,2,24,109,88,2,
   141,84,2,208,1,96,172,68,2,
   174,68,2,76,64,2
```

For systems running HEXDOS, lines 2 and 5 should be changed to:

```
2 POKE240,112:POKE241,A: REM
  SET UP USR(X)
5 A=PEEK(B+68);POKEB+68,1:Y=
  USR(-7):POKEB+68,A
```

For systems running OS-65D, lines 1 and 2 should be changed to:

```
1 RESTORE:A=PEEK(8960)-1:POKE
  8960,A:B=A*256:POKE133,A
2 POKE574,112:POKE575,A: REM
  SET UP USR(X)
```

END.

PROGRAM LISTING

```
0240  CA        DELAY 1  DEX
  41  D0 FD              BNE  DELAY 1
  43  A2 32              LDX  #32         Dissolve Rate
  45  88                 DEY
  46  D0 F8              BNE  DELAY 1
  48  A9 80              LDA  #80         Begin Scrn Mem (HB)
  4A  8D 55 02           STA  0255
  4D  A9 00              LDA  #00         Character Code
  4F  A0 98              LDY  #98         End Scrn Mem  (HB)
0251  A2 00     LINE 2   LDX  #00
  53  9D 00 80  LINE 1   STA  SCRNMEM + X
  56  E8                 INX
  57  E0 20              CPX  #20         Line Length
  59  D0 F8              BNE  LINE 1
  5B  EE 55 02           INC  0255        Incr SCRNMEM (HB)
  5E  CC 55 02           CPY  0255
0261  D0 EE              BNE  LINE 2
  63  AD 54 02           LDA  0254
  66  18                 CLC
  67  6D 58 02           ADC  0258
  6A  8D 54 02           STA  0254        Incr SCRNMEM (LB)
  6D  D0 01              BNE  LINE 3
  6F  60                 RTS
0270  AC 44 02  LINE 3   LDY 0244
  73  AE 44 02           LDX 0244
  76  4C 40 02           JMP DELAY 1
```

★ ★ ★

OSI KEYBOARD

By: John Whitehead
17 Frudal Crescent
Knoxfield 3180
Australia

The OSI keyboard is in the form of a matrix. It is accessed by sending data between 0 and 255 to its memory location of 57088 (HEX$DF00) and then reading that location to detect which key or keys are pressed. Due to incomplete address decoding, it actually covers memory 57088 to 57343.

For normal text use, a machine code routine in the MONITOR ROM at $FD00 takes care of keyboard scanning and decoding. This routine leaves the ASCII value of the key pressed in the 6502 accumulator. This routine can be called from BASIC with:

```
POKE 11,0 : POKE 12,253 :
     REM 253=$FD
X=USR(X) : A = PEEK (531) :
     PRINT CHR$(A)
```

The hardware for the Superboard C1P is different from the C4P. The C4P drives one row high at a time where the C1P drives one row low at a time. This makes the PEEK and POKE values different. Both are shown in the chart but examples are for C1P.

For special use, such as game movement keys or joysticks, a simple BASIC or M/CODE routine can detect keypresses.

To detect a single key, e.g., the space bar, look up the row and column values for "SPACE" in the chart and use them in the program below. The Ctrl C routine has to be turned off as it will POKE rows zero and two, which could give wrong column values. If you only want to detect keys in row zero, you can leave Ctrl C turned on and just PEEK the columns. The Ctrl C routine is at $FF9B. You could disassemble this to see how it works.

```
10 KEY=57088
20 POKE530,1 :REM TURN OFF
   CTRL C
30 POKE KEY,253 :REM DRIVE
   ROW TWO LOW
40 PRESS = PEEK(KEY) : PRINT
   PRESS: REM GET COLUMN VALUE
50 IF PRESS = 239 THEN PRINT
   "SPACE BAR" : POKE530,0 :
   STOP
60 REM 239 = COLUMN FOUR
   DRIVEN LOW BY CONNECTING
   IT TO ROW TWO
70 GOTO30
```

To detect two or more keys pressed together, a LOGICAL AND is performed on the row and column values from the chart, e.g., for Ctrl Z, drive rows 0 and 1 low and look at columns 5 and 6. To calculate a LOGICAL AND, the values need to be converted to binary. The LOGICAL AND of 0 and 0 = 0, 0 AND 1 = 0, 1 AND 1 = 1.

```
THE POKE VALUE IS:-
CTRL = 254 = $FE = %1111 1110
   Z = 251 = $FB = %1111 1011
                   ----------
                    1111 1010
                 =  $FA = 250
```

THE DATA SYSTEM

- Stored Report Formats
- Stored Jobs, Formats, Calcs.
- Multiple Condition Reports
- Multiple File Reports
- Calc. Rules Massage Data
- Up to 100 Fields Per Record
- User Designed Entry/Edit Screens
- Powerful Editor
- Merges - Append, Overlay, Match
- Posting - Batch Input
- Nested Sorts - 6 Deep
- Abundant Utilities

HARDWARE REQUIREMENTS: 48K OSI, Hard Disk, serial system, OS-65U 1.42 or Later; Space required: 1.3 megabytes for programs and data.

PRICE: $650.00 (User Manual $35.00, credited towards TDS purchase). Michigan residents add 4% sales tax. 30 day free trial, if not satisfied, full refund upon return.

TIME & TASK PLANNER

30 DAY FREE TRIAL — IF NOT SATISFIED, FULL REFUND UPON RETURN

- "Daily Appointment Schedule"
- "Future Planning List" - sorted
- "To Do List" - by rank or date
- Work Sheets for all Aspects
- Year & Month Printed Calendar
- Transfers to Daily Schedule

A SIMPLE BUT POWERFUL TOOL FOR SUCCESS

HARDWARE: 48K OSI, 8" floppy or hard disk, serial terminal system, OS-65U v. 1.3 or later.

PRICE: $300.00 (User Manual, $25.00, credited toward TTP purchase). Michigan residents add 4% sales tax.

FINANCIAL PLANNER

- Loan/Annuity Analysis
- Annuity 'Due' Analysis
- Present/Future Value Analysis
- Sinking Fund Analysis
- Amortization Schedules
- Interest Conversions

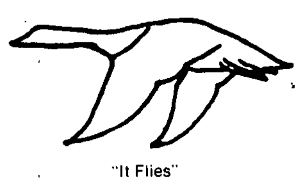
HARDWARE REQUIREMENTS: 48K OSI, 8" floppy or hard disk, serial terminal system, OS-65U v. 1.2 or later.

PRICE: $300.00 (User Manual, $25.00, credited toward Planner purchase). Michigan residents add 4% sales tax.

DEALERS: Your Inquiries Most Welcome

GANDER SOFTWARE, Ltd.

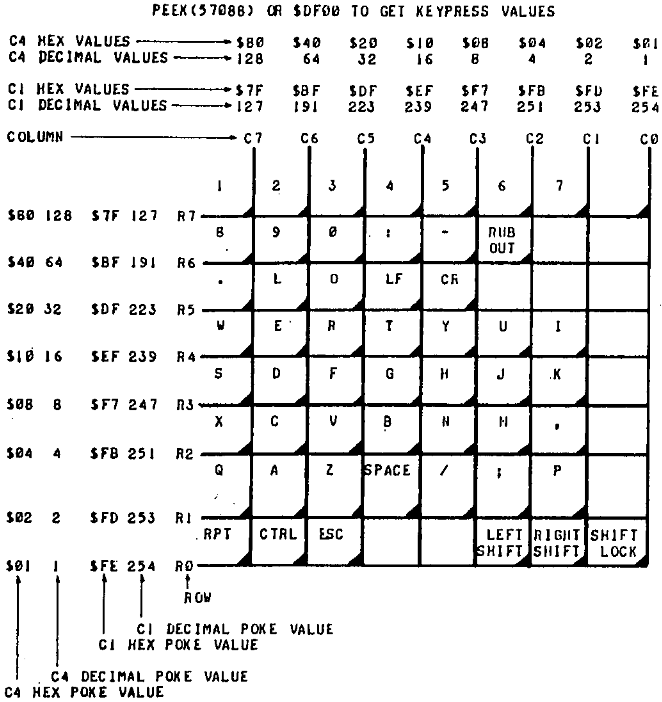
3223 Bross Road
"The Ponds"
Hastings, MI 49058
(616) 945-2821



"It Flies"

FROM THE FOLKS WHO BROUGHT YOU:
All This
THERE IS MORE COMING SOON:
Program Generator for TDS
Proposal Planner
Time and Billing A/R

```
THE PEEK VALUE IS:-
CTRL = 191 = $BF = %1011 1111
   Z = 223 = $DF = %1101 1111
                   ---------
                   1001 1111
                 = $9F = 159
```



BASIC will do the above calculation for you as shown below.

```
100 KEY=57088
120 POKE530,1
130 POKEKEY,254AND253
140 PRESS=PEEK(KEY):PRINTPRESS
150 IF PRESS =(191AND223) THEN
    PRINT"CTRL Z":POKE530,0:
    STOP
170 GOTO130
```

If you run the above, you may find you can't escape. This is because you have three keys pressed, one in row one, two in row zero, and the column value is not 159. This can be fixed by changing line 150 to:

```
IF PRESS = (191AND223) OR
   PRESS = (191AND223AND254)
   THEN....
```

```
 0 TRY RUNNING THIS LITTLE
   PROGRAM USING ROM BASIC
 1 :
 10 REM    PRINTING IN ANY
           ORDER
 20 POKE4,194:POKE5,165
 30 REM    SELECTED LINES FOR
           SAVING OR
 40 LIST50:LIST30:LIST10
 50 REM    THIS CAN BE USED TO
           LIST
 60 POKE4,195:POKE5,168
OK
```

★ ★ ★

BEGINNER'S CORNER

By: L. Z. Jankowski
Otaio Rd 1, Timaru
New Zealand

DEBUGGING & TESTING OF PROGRAMS

PART II

PROGRAMMING MISCONCEPTIONS

Effective debugging is impossible if the programmer is wrong about how the programming language works or if there are misconceptions about programming logic.

What will be the output in response to this line:

```
10 IF A=1 or 0 THEN PRINT
"Done"
```

Change the "OR 0" to OR A=0" and compare the new output with the old. Both expressions are syntactically correct. Which is the "right" one would depend on what the programmer was trying to do.

Will BASIC accept this line, and if so why?

```
10 X=4: Y=4: IF X=Y-1
   THEN PRINT "TRUE",X,Y
```

Even this is valid: A=4=B=C=D=E. But only the first "=" sign actually assigns a value to a variable. The "=" signs that follow the first one are interpreted by BASIC as, "compare and evaluate the expression as either TRUE or FALSE".

Try this program:

```
10 Y=9: DEF FN A(X) = X*X
20 PRINT FN A(3)=Y
```

BASIC prints the value "-1" because it is "TRUE" to say that "FN A(3)" has the same value as "Y".

How many times will this loop be executed? Answer the question, then test your answer.

```
10 FOR C=4 TO -1: PRINT C:
   NEXT
```

DIM N$(20) declares an array with 20 entries .....plus one for N$(0), makes 21!

Will BASIC accept the next line?

```
100 IF X=0 THEN: FOR A=1 TO
    10: NEXT
```

The next program is an example of recursion - line 100 calls itself. (Dictionary definition: Recursion - see Recursion). The program prints numbers from the Fibonacci Sequence, used in the study of phyllotaxy and organic growth, amongst other things.

```
30 PRINT "1 , 1 ,";:
   K=2:P=1:C=1
100 N=P+C: PRINT N ",";:K=K+1:
    P=C:C=N:GOSUB 100
```

Variable "K" in the program counts the number of elements printed. The program stops with "OM ERROR" when 28 numbers have been printed from the sequence. There is plenty of memory free. What BASIC is trying to indicate with "OM ERROR" is that the stack is full - too many GOSUB calls. Change the program to avoid this problem. (Hint: use GOTO). The program will now halt because of "OV ERROR" - the numbers have gotten too big. What is the value of the 185th member of the Fibonacci Sequence?

Programmers sometimes write this: "IF F THEN ....", to save typing "IF F<>0 THEN ....".

What will BASIC do to this line?:

```
10 IF Q$="Y" OR F OR X>C-1
   THEN 480
```

It is important to clear up misconceptions about programming and the programming language being used.

TRACE

The "spy", the "dump" and the "trap" are powerful debugging techniques. There is also the "trace". Some trace programs are very fast; others are very slow. There are three good trace programs available to OSI disk users. There is the standard TRACE utility as offered by DOS 3.3 and 3.2. Program output and line numbers are written continuously to the screen at high speed. There is no line-feed, carriage return between the line number and program output. For those without a bionic eye, use CTRL-S to stop the trace in order to examine the screen, press any key to continue program execution.

The "HOOK" trace is more useful. It works exactly like the OSI trace but uses the "T*" command as an on/off toggle and can be part of a BASIC program. It is possible, therefore, to selectively trace any part of a BASIC program.

Another useful trace program, also by Rick Trethewey, is found on pg. 78 in the MICRO OSI book. The program actually lists the line of BASIC being traced. What is more, the values of all variables are also optionally printed. The trace waits for a keypress before continuing.

Tape users also have access to an excellent trace, written by M. Piot; see MICRO, July '81 issue.

BLOCK EXECUTION

No, not a punishment for errant programmers, but another debugging and testing tool. It is possible to run a program from any line number, e.g., RUN 285. Using the word RUN clears all variables back to zero or null, but using GOTO and GOSUB does not clear variables. A block of code can be easily tested from a GOTO or GOSUB in Immediate Mode type "CLEAR", and then define the required block variables. Now type "GOTO" and the line-number, to execute the block of code. If the program bug is in the block, then it is effectively isolated to a specific piece of code. If the bug is not present, then that information is equally valid. Remember, the STOP command can be inserted into the block at will, to narrow down the search still further.

TESTING

Testing a program is an attempt to prove that it will work with all valid data and that it will cope with most (all?!) invalid data. Debugging a program merely removes all visible errors.

An excellent way of testing code is to see whether it does "nothing". In other words, the program should not crash because it is working with null data. This is one end-condition. Test the program with the other end-condition. Test the program with valid data. What does the program do with invalid data, e.g. "Ø", negative values, null input? Is invalid data creeping in, inadvertently causing the program to crash? If so, where? Testing with end-conditions is not exhaustive, so that elusive bug could still remain. Test with different end conditions. This is always a good way of cleaning up a "BS ERROR" (array bad subscript error). The program has attempted to reference an array value beyond the array definition. The solution to the problem is not to extend the array definition!

The following program will calculate and print the logarithm table for any base. "RUN 2Ø" prints the whole table. "RUN" confines "NUMBER" to integer values only. The program will crash for certain values. Insert a line 35 which checks that the chosen values of N, F and T are in the correct range.

```
1Ø I=-1
2Ø INPUT "LOGARITHM BASE ";N:
   PRINT
3Ø INPUT "FROM value ";F:
   INPUT "TO value ";T
4Ø PRINT: PRINT: PRINT "BASE
   LOG NUMBER": PRINT
5Ø :
1ØØ FOR C=F TO T
2ØØ :  Al=LOG(C) / LOG(N)
3ØØ :  IF Al>INT(Al) AND I
   THEN 5ØØ
4ØØ :  PRINT N" ^ " Al " ="
    TAB(13) C
5ØØ NEXT
6ØØ PRINT: PRINT "DONE!"
```

BASIC ERROR CODES

Responding correctly to BASIC error codes is important. For example, in response to an "NF ERROR" (NEXT without a matching FOR) it is important to match up correctly, "FOR" statements with "NEXT" statements. Do NOT use a "GOTO" to get out of trouble! But sometimes the BASIC error message can be misleading, or just plain uninformative as to where the real error is. For example, the program below stops in line 1Ø. Where is the real error?

```
1Ø FOR C=1 TO 1Ø: READ Y:
   POKE 54954,Y: NEXT

5ØØ DATA 24Ø, 2ØØ, 234, 196,
    171, 245, 254, 256, 243,
    238
```

Or, how about this one:

```
1Ø PRINT CHR(95)
```

BASIC reports "BS ERROR" but all that's missing is "$" after "CHR"! If the number in brackets was less than 1Ø then no error would be reported. BASIC will accept undeclared arrays if the dimension is less than 1Ø.

EUREKA!

The bug has been found, - Eureka! But before leaping out of the bath to race down the road, it is of value to consider more than just one's unclothed state. Is the bug found the right one?! Is there another bug immediately following the discovered one? Could it be that the discovered bug is repeated elsewhere in the program? No matter where the bug was, it is worth testing the WHOLE program again. It is not unknown for a solution to an error to introduce bugs of its own. Finally, in many instances a second opinion is worth seeking. Working with another programmer is an excellent idea. Keep a notebook of bugs found and what the solution was.

NOT MY FAULT!

It is a natural reaction when the bug strikes to blame the computer, the dog, the weather or even the BASIC! Although the reaction is best curbed severely, on occasion criticism of BASIC is justified. The infamous garbage-collect bug of OSI ROM BASIC is one such example. Another infuriating problem is wrong answers to calculations. The answer is obviously and simply "1" but BASIC comes up with ".999999999"! These are BASIC internal, base-two, floating point calculation algorithm errors - I think!

Here is another one from OS-65D 3.3. Run this program.

```
1Ø PRINT& (Ø,Ø)!(24) & (4Ø,15)
   "A" & (4Ø,15)!(33):
   INPUT Y$
2Ø PRINT& (Ø,18) "X" Y$ "Y"
   ASC(Y$)
```

Program output will be "XAY 65". Now change the '"A"' to '" "'. Program output will be 'X Y 32'. Wrong! Program output is "XY" followed by "FC ERROR". "Y$", in spite of reading a blank off the screen, has had its value changed from a blank to a null. Consequently, ASC(Ø) generates the error message. Nothing in the manual 'bout that!

THE POOR PROGRAMMER

Debugging is hard work, demanding time, patience, intuition, information and scientific analysis. Employ the Golden Rule, "What is this segment of code supposed to do?". When debugging, make changes in the program ONE at a time. Be prepared to re-write the code. Don't believe in magic, but be prepared for the bizarre - and not just with OSI stuff either!

NOTES ON WP65Ø2 V1.3, 5.25"

By: Paul Chidley
Courtesy of TOSIE
Toronto Ohio Scientific Idea Exchange
P. O. Box 29
Streetsville, Ont.
Canada L5M 2B7

Below are some of my notes on WP I would like to share with you.

What happens when you hit "D"? The floppy boot routine in ROM loads track Ø into RAM starting at $22ØØ for 8 pages. (1 OSI page = 256 bytes, therefore, 8 pages = 2K bytes.) The CPU then jumps to $22ØØ where the code from track Ø then proceeds to load track 1. When finished its initialization routines, it then does a JMP to $2ADE. This is where you will find the Load Common Subroutine that loads 5 tracks starting with track 7 into RAM starting at location $Ø2ØØ. The CPU then jumps to $Ø222 (NOT $Ø2ØØ), which then jumps to $1FF7, which then continues on into WP65Ø2.

Now that you know WP's entry point, hit break, then do a GO $Ø222 and you're back in WP. This is very handy if you should want to change some value in the operating system (such as step speed at $26A3) and then re-enter WP without rebooting.

Have you ever modified an I/O driver on track 0 only to boot WP6502 to find it doesn't work? Well, I have already told you why. The load routine loads 5 tracks, starting at track 7, into RAM starting at location $0200. If you examine your disk you will see that track 11 is 8 pages long. When all 8 pages of track 11 are loaded, all of track 0 has been overwritten, including much of the operating system. The memory layout is illustrated below. The problem and the fix are both really quite simple. Somewhere along the way (probably when they converted the 8" version to 5") track 11 was put on the disk as being 8 pages long. This was a snapshot of what their system looked like when they did it, and when you boot, it overwrites your systems values. To fix this major problem, simply call track 11 into memory and save it back using 1 as the number of pages. Then when you boot, it will load from $2200 to $22FF and leave your tables and drivers alone.

```
TRK#/Addresses in HEX
---------------------
07 0200 - 09FF
08 0A00 - 11FF
09 1200 - 19FF
10 1A00 - 21FF
11 2200 - 29FF Should be
          2200 - 22FF
```

Have your ever booted your copy only to get a "Pr.Check" message? If you have, the odds are that you have been modifying the hardware. On boot up, WP expects to see a daisy wheel printer at $F500. If you can afford WP6502, then surely you can afford a daisy wheel, right? Well, I'm sure somebody has one somewhere. The problem is that if you put ROM or anything else at $F500, then WP may think that what it sees is the daisy wheel's status word. It, therefore, prints the message to tell you to replace the paper in the printer you haven't got. You could also see this error if you have very noisy lines, what it wants to see is $F5, if it doesn't, out comes the error. It assumes that if it sees $F5, then you haven't got a daisy wheel. The subroutine that actually prints the message is at $19EF. The location we are interested in is $1992. To solve the problem change the contents of $1992 from $F5 to what your system contains at that location. A better way is to change the code so that it always branches, but I won't show you that one, try it, just look in the area of $1992.

Try looking at $2336, I think it is WP's keyboard routine.

Author's note: Make sure you also read the article "65D V3.3 Bug (Oct. page 8 & Nov. page 7)," my copy of WP6502 also suffered from the same problem. I recently got 8" drives as well as 5" working on my system so I have started disassembling WP6502 V1.3A. This version fixed many of the problems with V1.3 but it was never released on 5", I want to disassemble it both to learn more and to convert it to 5". If you are interested, drop me a line.

★

USR (X)(Y)(Z)(1)(2)(3)

By: Earl Morris
3200 Washington
Midland, MI 48640

BASIC's USR function normally has only one input variable. By calling the proper routines in a Machine Language program, additional variables can be called from BASIC. The following program allows a flexible number of input variables. After reading a variable, the program looks for another "(" in the BASIC line. If found, another variable is read in. The program as written adds up the input variables and returns the sum to BASIC. Integer addition is used so that decimals will be truncated. The input can be simple numbers, letter variables, expressions or functions. The example BASIC program uses from 2 to 5 inputs to the same USR function.

The program as written was assembled at $8000 and runs with 65D 3.2. Changes are given for ROM BASIC. If any other readers have written interesting USR functions, how about sending them in to PEEK(65).

Sample BASIC Program

```
10 POKE 574,0!POKE575,128 !REM FOR DISK
                                  BASIC
20 A=2:B=-10!C=100
30 PRINT USR (2)(2)
40 PRINT USR (A)(B)(C)
50 PRINT USR (A+B)(C/2)(10*2)
60 PRINT USR (10)(10)(10)(10)(10)
```

Sample Output

```
OK
RUN
 4
 92
 62
 50

OK
```

```
10;         USR (X)(Y)(Z)...
20;            FOR DISK BASIC - ROM CHANGES BELOW
30;
40;         RETURNS SUM OF INPUTS ( VARIABLE NUMBER)
50;         TO SET UP USR VECTOR
60;          DISK BASIC  POKE 574,0!POKE 575,128
70;
80         *=$8000
90         LDA #$00
100        STA SUML      CLEAR SUM LO BYTE
110        STA SUMH      CLEAR SUM HI BYTE
120GO      JSR $1056     FLOATING TO BINARY
130        CLC           PERPARE TO ADD
140        LDA $B2       GET INPUT VALUE LO BYTE
150        ADC SUML      ADD TO SUM
160        STA SUML
170        LDA $B1       GET INPUT VALUE HI BYTE
180        ADC SUMH      ADD TO SUM
190        STA SUMH
200;
210;       NOW CHECK TO SEE IF ANY MORE VARIABLES
220        LDY #$00
230        LDA ($C7),Y   GET NEXT CHARACTER IN BASIC LIN
240        CMP #'(       CHECK FOR LEFT PAREN
250        BNE DONE
260        JSR $0CCD     GET NEXT VALUE FROM BASIC
270        JMP GO        REPEAT PROCESS
280;
290;       LOAD REGISTERS WITH SUM AND RTS TO BASIC
300DONE    LDA SUMH
310        LDY SUML
320        JMP $1218      PASS VALUE TO BASIC AND RETURN
330SUML    .BYTE $00
340SUMH    .BYTE $00
350;
360;        CHANGES FOR ROM BASIC
370;        POKE 11,0!POKE 12,128
380;        $1056 TO $AE05
390;        $B2 TO $AF
400;        $B1 TO $AE
410;        ($C7) TO ($C3)
420;        $0CCD TO $AAAD
430;        $1218 TO $AFC1
```

★ ★

F D U M P

By: Roger Clegg
Data Products Maintenance Corp.
9460 Telstar, El Monte, CA 91731

My FDUMP is much slower than OSI's machine-code one, but much smaller, five times faster than their old BASIC dump, and can do both ASCII and Hex.

```
1 REM **********************   F D U M P   ******************************
2 :
10 C=0: I=0: L=31: U=127: CR=13: P=46: F=15: S=16: CB=9889: CC=15006
20 RAM=30000: Q=256: Q2=65536: Q3=16777216: CR$=CHR$(13)
30 X=PEEK(9832): DV$=CHR$(65+X): IF X=128 THEN DV$="E"
40 PRINT: INPUT"DEVICE  ";D$: IF D$="" THEN D$=DV$
50 PRINT TAB(22)"<RETURN> = #KEY";CR$;: INPUT"FILE    ";FILE$
60 PW$="ANAN": IF FILE$="" THEN FILE$="#KEY"
70 DEV D$: CLOSE: FLAG 9: OPEN FILE$,PW$,1: FLAG 10
80 FA=Q*PEEK(9907)+Q2*PEEK(9908)+Q3*PEEK(9909)+16: REM File address
90 FS=Q*PEEK(9910)+Q2*PEEK(9911)+Q3*PEEK(9912)-16: REM File size
95 :
100 INPUT"PORT    ";D
110 PRINT TAB(22)"<RETURN> = No"CR$;: INPUT"HEX DUMP";R$
120 H=0: IF R$="Y" OR R$="H" THEN H=-1
130 IF H=0 THEN 180
140 DIM HX$(15),HEX$(255)
150 FOR I=0 TO 15: HX$(I)=MID$("0123456789ABCDEF",I+1,1): NEXT
160 FOR I=0 TO 15: FOR J=0 TO 15: HEX$(I*S+J)=HX$(I)+HX$(J): NEXT: NEXT
170 CPL=16: GOTO 200
180 PRINT TAB(22)"<RETURN> = 64"CR$;: INPUT"CHARS PER LINE";CPL
190 IF CPL<1 OR CPL>69 THEN CPL=64
200 ST=-(LEFT$(FILE$,1)="#")
210 IF ST=1 THEN PRINT TAB(22)"(or # sign and record number)"CR$;
220 PRINT TAB(15)ST;CR$;: INPUT"STARTING INDEX";R$: PRINT
230 IF ST=1 AND LEFT$(R$,1)="#"THEN IX=Q*(VAL(MID$(R$,2))-1)+1: GOTO 260
240 IX=ST: IF R$<>"" THEN IX=VAL(R$): IF IX<0 OR IX>=FS THEN IX=ST
250 IF ST=1 AND IX<>0 AND CPL=64 THEN IX=64*INT(IX/64)+1
260 IF D<2 THEN PRINT" "FILE$" OPEN FOR ALTERATIONS AS FILE #1": PRINT
270 IF D>1 THEN PRINT#D,TAB(28)"DUMP OF "FILE$: PRINT#D: PRINT#D
280 :
300 FLAG 25: POKE 8778,192: POKE 8779,36
310 POKE 9432,243: POKE 9433,40: POKE 9435,232: POKE 9436,40
320 DA=FA+IX: REM Disk address
330 NB=CPL*INT(16000/CPL): IF NB>FS-IX THEN NB=FS-IX: REM Number bytes
340 DH=INT(DA/Q3): X=DA-DH*Q3: DM=INT(X/Q2): X=X-DM*Q2: DL=INT(X/Q)
350 POKE CB+1,X-DL*Q: POKE CB+2,DL: POKE CB+3,DM: POKE CB+4,DH
360 POKE CB+5,NB AND 255: POKE CB+6,NB/Q
370 POKE CB+7,48: POKE CB+8,117: REM Transfer to RAM at 30000
380 ERR=USR(0): IF ERR THEN PRINT"DEV "D$" ERROR"ERR"IN 380": GOTO 460
390 :
400 FOR J=RAM TO RAM+NB-1 STEP CPL: PRINT CR$;
410 IF PEEK(CC) THEN POKE CC,0: INPUT" Continue";R$: IF R$="N" THEN 460
420 PRINT#D,J+IX-RAM TAB(10)
430 IF H THEN GOSUB 600: NEXT J: GOTO 450
440 GOSUB 500: PRINT#D: NEXT J
450 IX=IX+NB: IF IX<FS THEN 300
460 POKE 8778,208: POKE 8779,16: FLAG 26
470 DEV DV$: END
480 :
490 REM   ASCII DUMP
495 :
500 FOR I=J TO J+CPL-1: C=PEEK(I)
510 IF C=0 THEN PRINT#D,"_";: NEXT: RETURN
520 IF C>L AND C<U THEN PRINT#D,CHR$(C);: NEXT: RETURN
530 IF C=CR THEN PRINT#D,"#";: NEXT: RETURN
540 PRINT#D,"@";: NEXT: RETURN
570 :
580 REM   HEX DUMP
590 :
600 FOR I=J TO J+F: PRINT#D,HEX$(PEEK(I))" ";: NEXT: PRINT#D,"|";
610 FOR I=J TO J+F: C=PEEK(I): IF C<=L OR C>=U THEN C=P
620 PRINT#D,CHR$(C);: NEXT: PRINT#D,"|": RETURN
630 :
640 :
50000 FLAG 10: ERR=PEEK(10226)
50010 IF ERR=130 OR ERR=131 THEN INPUT"PASSWORD";PW$: GOTO 70
50020 IF ERR=128 THEN PRINT: PRINT"FILE NOT FOUND": GOTO 40
50030 PRINT"DEV "D$" ERROR"ERR"IN 70": DEV DV$: END
```

★ ★

BETA/65
A REVIEW

By: D. G. Johansen
P. O. Box 252
La Honda, CA 94020

PRODUCT DESCRIPTION

BETA/65 is a recently developed high-level language for the 6502 microprocessor. During formulation of BETA/65 it was recognized that available languages for microprocessors were developed over two decades ago for a different computing environment and user community. Systems designed in that era were optimized for mainframe computers and used mainly for numerical processing applications.

Today's computing applications are more diverse compared with earlier times and the average user is less likely to be a computer specialist. Languages of the 1960s are capable of meeting the needs of today's user. However, this is done in an ad hoc manner causing systems to be more complex when users are demanding simplicity.

The proliferation of computers means that the potential user base for a language is far greater than in the past. For this reason, upgrading the quality of programming systems is regarded as a very necessary and worthwhile enterprise.

ADVANTAGES OF BETA/65

BETA/65 was developed to test several programming concepts which have surfaced in recent years. Each of the following advantages may appear in one or more other programming systems. However, BETA/65 is the only system integrating these features into a common package.

Interpreter Based - Interpretive systems need only one file for program representation. Compiler-based systems such as FORTRAN, PASCAL, and C require two (or more) files and this is a source of complexity for these systems. Interpreter-based systems are interactive because the source-to-object compile step is eliminated. The stigma of slow run time associated with interpreters has been largely eliminated by use of byte-codes designed for high-speed interpretation.

Direct Notation - An APL-like

notation is used for programming BETA/65 expressions. APL has not appeared on most micro-based systems due to incompatibility with the ASCII character set found on most keyboards. BETA/65 substitutes ASCII names familiar to BASIC programmers and brings the advantages of direct (APL-like) notation to the standard keyboard.

Extensions - Direct notation allows new functions to be easily assimilated into the language. This is not readily done with BASIC due to use of algebraic notation, which has complicated precedence rules for function execution. With direct notation, only four function types are used. These are sufficient to describe any mathematical expression in a natural and easily verifiable form. New functions, for specialized applications, may be added to the primitive instruction set by the user.

Mixed-Precision Arithmetic - All functions operate on mixed-precision data from one to fifteen bytes for either argument. Use of mixed precision virtually eliminates the scaling problem and frees the programmer for more productive work. Also, mixed precision allows separate programs, working at different precision levels, to easily exchange data. This simplifies programming in an integrated environment.

APPLICATIONS

The author of BETA/65 has over 25 years experience designing navigation and control systems for aircraft and spacecraft. These applications demand reliable and accurate operation, and must present a well-conceived user interface. Comparable applications include peripheral control, graphics, tele-communications, and control processing.

Peripheral Control - Microprocessors now appear in computer peripherals (e.g., printers, display terminals), and stand-alone devices, such as copying machines and consumer appliances. Due to its compact size (less than 20K), BETA/65 may be used for on-site software development, staying with the target machine through the life cycle. This greatly simplifies software development and end-use maintenance.

Control Processing - BETA/65 was initially formulated for control processing applications. Machine connections are built-in, allowing access to high-speed machine code and interface ports. The LINK function allows user calls to machine code by name or address. In addition to PEEK and POKE, DPEEK and DPOKE are supported. The latter two functions allow two-byte (pointer) modification with one instruction. Concurrent entry allows user input without interruption of the running program.

Graphics - Dot-matrix printers and bit-mapped video screens offer new media for graphics expression. BETA/65 provides video windows for full use of display devices. Logical functions are provided, allowing bit-level manipulation. String functions support text-oriented applications. Typical graphics applications range from custom logo and type-face creation, to quick-look data display.

★

OS-65U DATA FILES AND OTHER MYSTERIES: FEAR AND LOATHING GUIDE

PART II

By: Rick Trethewey
8 Duran Court
Pacifica, CA 94044

With OS-65U's INDEX command, you can construct random access files with any record length you choose, from one to the capacity of the disk, and the fields within these records can be stored with a precise offset from the beginning of the record, thus allowing random access at the field level. Let's look at an example of a record in a phone number file:

Field	Name of Field	Length
1	NAME	35
2	PHONE NUMBER	12

In this file, the true record length is 49 bytes because we have to allocate one extra byte for the carriage return that terminates the entry in each field. For any data file, the true record length can be calculated as the sum of the lengths of all of the fields in each record plus the number of fields. Of course, the addition of the extra byte for the carriage return can be assumed to have been included in the field lengths listed above and you can save yourself a step. For now, let's assume the above figures do include one byte for the carriage return.

Setting the INDEX command to point to the start of any record is done with the formula/command (forgive the PASCAL-like variables);

```
INDEX<CN> = (RecordNumber - 1)
    * RecordLength
```

The reason we subtract one from the record number is because most people refer to the first record in a file as record number 1. Gaining access to individual fields within records requires a slightly more complex formula. To determine the value to give to INDEX in order to point to field number "FieldNumber" in record number "RecordNumber", we could use this subroutine;

```
RecordIndex = (RecordNumber-1)
  * RecordLength

FieldIndex = RecordIndex:
  IF FieldNumber = 1 THEN
  RETURN

FOR K = 1 to FieldNumber-1

FieldIndex = FieldIndex +
  FieldLength(K)

NEXT K: RETURN
```

Of course, for most applications, the offset to each field in the record is calculated before the program needs to use the data file. Then, whenever a field is needed, the only math required is to add this offset to the value resulting from the calculation to find the start of the desired record, thus making for a faster and more compact program.

The value of being able to directly access individual fields within any record in your data file cannot be overestimated. Think of it for a moment. If the information you need is in field number 7 of some record in your data file, but you don't know which record, a simple brute force scan of your data file would be many times faster if your program could simply ignore the first six fields in each record it had to search. Another feature of OS-65U makes this ability even more valuable, and that is the INDEX function. The INDEX function returns the value of the INDEX pointer for any data channel you have open. The syntax of the INDEX function is;

```
X = INDEX(CN)
```

where "CN" is the channel number. When executed, "X" will hold the current INDEX pointer to the file in question. The INDEX function is most often used to determine the record number being pointed to during some file operation in which the pointer is not under the explicit control of the program. This is the case when the FIND command is used.

The FIND command under OS-65U does a brute force search of a data file, looking for a string specified by the user. This search begins at the current INDEX pointer position and proceeds until a match is found or the end of the data file is encountered. The syntax of the FIND command is;

```
FIND "string", CN
```

where "string" is the string being searched for, and "CN" is the channel number being used to access the data file. Unlike its OS-65D counterpart, FIND under OS-65U resets the INDEX pointer to the start of the "found" string if a match is found. Thus, the INDEX function would return a value pointing to that string, giving us the ability to calculate the record number in which the match was found. The required calculation would be something like;

```
RecordIndex = INDEX(CN)
RecordNumber = INT(Record
  Index/RecordLength) +1
```

Here again, the "1" is needed to reflect the cardinal numbering of records. The user should note that an additional calculation may be needed in order to determine if the match was found within the expected field number. Such a calculation would be needed if the string being searched for might occur in more than one field. Subtracting the RecordIndex from the current INDEX (CN) yields the offset from the start of the record to the string located by FIND. A sequential search of the field offsets will let you deduce the field in which the match was found.

The FIND command brings us to another feature of OS-65U, the FLAG command. The FLAG command under OS-65U replaces many of the POKEs that we use under OS-65D to turn features on and off. The FLAG command syntax is;

```
FLAG xx
```

where "xx" is a number, usually between 1 and 33 under most versions of 65U. In most cases, the command "FLAG x" will enable a feature and "FLAG x+1" will disable a feature. There are three sets of FLAGs that are especially useful for dealing with data files. The first is FLAG 9, which is much like the TRAP command under OS-65D V3.3, except that this feature only traps disk errors, and when a trap occurs, the program is sent to line #50000. There are some PEEKs available to determine exactly which disk error was encountered and the line number in the program where the error occurred. FLAG 10 disables FLAG 9 and causes a program to stop after any disk error. The next FLAG to consider is FLAG 11. FLAG 11 prevents OS-65U from writing the leading spaces generated by BASIC when it PRINTs a positive number. This can save you quite a bit of space in a large data file, and since it can only cause a problem if you really try, I always enable it. FLAG 12 allows leading spaces to be sent to the data file. Finally, we come to the FLAG which is essential when using the FIND command, and that is FLAG 5. FLAG 5 sets the value of the INDEX pointer to 1E9 if the search fails to find a match. FLAG 6 forces a normal disk error if the end of the data file is reached (note that such an error can still be trapped by FLAG 9 even if FLAG 6 is executed).

Now that we have all the tools that give us precise control over where we will read or write in our data file, we need the actual commands to perform these operations. As I noted above, we use INPUT and PRINT, but in a special format. To read information from a data file opened using channel #CN, we would use;

```
INPUT %CN, variable
```

where "variable" is the variable in which to store the information read from the data file. Likewise, to write information, we would use;

```
PRINT %CN, variable
```

Don't forget that with random access files, an INDEX command will likely precede any INPUT or PRINT command.

The formulas and calculations presented above that determine the INDEX needed to use a data file, all have a significant flaw. That flaw is that they require the program to know the complete structure of the data file before the program is written. On the other hand, if we construct our data files in a uniform manner, we can write programs that can use all of our data files. The most common way of doing this is to store information about the features of the data file which change from one to the next at the very start of the data file. The term used to describe the area of the data file in which this information is stored is called the "header". As you might expect, even the structure of the header must be uniform from one file to the next if we are to succeed here. In 1979, Ohio Scientific developed a package of programs designed to generate random access files and they call it OS-DMS. OS-DMS is by far the most common data file structure in use on OSI systems today. While not without flaws, OS-DMS is a simple structure that is well-suited to use with OS-65U. With all of that in mind, let us examine how OS-DMS works.

To begin, think about the characteristics of random access files that vary from one to another. They are the record length, the number of fields in each record, the length of each field, and the name of each field. Therefore, all of those features must be stored in the header. But we need to know a bit more before we can use the data file. We also need to know how many records the data file can hold and we also need to know how many records have been stored in the file thus far, and this too is stored in the header. Finally, OS-DMS requires that the name and type of the file be included in the header. There are two types of OS-DMS data files, Master Files and Key Files. Master Files are random access files that hold all of the information needed by the applications, and Key Files hold information about the contents of individual fields within associated Master Files and pointers to the Master File. By convention, OS-DMS data file names are a full six characters long, with the first five being the name of the file and the last character being a number which determines the file type (0 = Master File, 1 through 7 = Key File). Therefore, "MAIL 0" is a Master File and "MAIL 1" is a Key File associated with "MAIL 0". Note that the numeric file name extension on Key File names does not determine the field number used by that Key File.

The structure of the header of an OS-DMS Master File is as follows;

CONTENTS	INDEX
5 character file name	0
2 character file type	6
End of Data File Index	9
Beginning of Data File Index	20
Record Length	31
Maximum Number of Records	42
Field Name(s), Field Length(s)	53

.
.
.
.

Rec. #1 Beginning of Data File

Note that beginning at INDEX 53, the field names and field lengths are stored sequentially. A typical subroutine to open an OS-DMS Master File might look like this;

```
1000 OPEN "MAIL 0","PASS",1
1010 INDEX<1>=0 : INPUT%1,
     NAME$
1020 INDEX<1>=6 : INPUT%1,
     TYPE
1030 INDEX<1>=9 : INPUT%1,
     EODF
1040 INDEX<1>=20: INPUT%1,
     BODF
1050 INDEX<1>=31: INPUT%1, RL
1060 INDEX<1>=42: INPUT%1, NR
1070 INDEX<1>=53: NF=0: I=0
1080 INPUT%1, F$,FL: NF=NF+1:
     IF INDEX(1)<BODF THEN
     1080
1090 DIM F$(NF), FL(NF),
     I(NF), A$(NF):INDEX<1>=53
1100 FOR K = 1 TO NF
1110 INPUT%1, F$(K), FL(K)
1120 I(K)=I: I=I+FL(K)
1130 NEXT K
1140 TN = INT((EODF-BODF)/RL)
```

Lines 1010 through 1060 retrieve the elementary information about the Master File. Line 1070 sets the channel INDEX to the start of the field names and lengths and also initializes the field number counter and a variable used to determine the field offsets discussed previously. Notice that the method used to determine the number of fields. That is, you read a field name/length pair, and increment the number of fields counter until the channel INDEX reaches the beginning of the data file. Line 1090 dimensions arrays to hold the field names, lengths, index offsets, and record contents. The FOR...NEXT loop from line 1100 through 1130 reads in the field name/length pairs and calculates the offset from the start of the record to each field and stores it in the array I(n). Finally, line 1140 calculates the number of records stored in the data file by dividing the difference between the end of the file and the start of the file by the record length.

Using this structure, a subroutine to retrieve a record might look like;

```
2000 RPTR = (RN-1) * RL: REM-
  Remember this formula?
2010 FOR K = 1 to NF
2020 INDEX<1> = RPTR+I(K):
  INPUT %1, A$(K)
2030 NEXT K: RETURN
```

This routine retrieves the entire record in the array A$ (X). Note that I put the calculation that determines the INDEX of the start of the record outside of the FOR...NEXT loop in order to speed it up. Changing INPUT to PRINT in line 2020 would write the array A$(X) as record number "RN". Look closely at line 2020 for a moment and see how the value in I(X) is used. Note that I(1) always equals zero since field number 1 is always at the start of the record. From there, I(2) = FL(1) and I(3)=I(2)+FL(2)... I(NF) =I(NF-1)+FL(NF-1). I have found this code to be fast and easy to follow in programs.

This brings us to Key files. What are Key files? Simple. Key files are files that are associated with Master files which contain the contents of a selected field from each record in the Master file, along with the INDEX to the start of the corresponding record in the Master file. OK, you say, but what good are they? I've asked myself that question a number of times before I realized what a big help they can be. OS-DMS allows up to seven Key files to be associated with each Master file. If you'll recall, the file name of every Master file ends with a "0". Key file names end with a digit from 1 to 7. This digit is only to differentiate between Key files and is not indicative of the field number that the Key file holds as a reference. The purpose of Key files is to allow quick scanning of selected fields within a Master file. Since Key files are sequential, they are very compact and thus they require a minimum of disk space. Their main claim to fame is their facility for being quickly sorted. For example, let's look again at our phone book data file. Occasionally, we may want the file dumped in alphabetical order. Other times, we may need to see the file listed by area code. While we could sort the entire Master file each time we need to access its contents in a special sequence, it is faster and easier to do so by using Key files. And it is not just the order of access that can be keyed upon. For example, one Key file may point to records in a customer list who buy one kind of product, and a separate Key file can point to customers who buy other products. OS-DMS Key file entries begin at an INDEX of 53 and are made up of two pieces of information. The first piece is the contents of the record in question preceded by a caret symbol (i.e., "↑"). The second piece is the INDEX to the start of the record. Since it is a sequential data file, however, special care has to be taken when this file is used. For example, if you want to edit an individual entry in the Key file, you have to read in the entry pair to be edited and also every subsequent entry to the end of the file. Further, once the entry is edited, it and all of those subsequent records must be re-written out to the file. Fortunately, there is little cause for editing individual entries. Indeed, doing so would be an exception to normal practice. Ordinarily, all updates to a Key file will involve a total rewriting of the entire file's contents, whether you are sorting the current contents of the Key file or reloading it to reflect new entries in the associated Master File.

Continued next month.

★