SOFTWARE & CORRECTION

Dear Jim,
Received: 77 Oct 28

My 650X subroutine OPLEGL, published in the Aug. '77 *DDJ* let one illegal opcode (9E) slip through its net. So there 105 illegals, not 104. I am therefore submitting a rewrite that fixes this error and also spruces things up some. So that OPLEGL will be more than an academic exercise in sorting, I have added a primitive debugger program that uses both OPLEGL and BYTNUM. I am also submitting a new byte-count-computing subroutine (NUMBYT) that uses utterly different logic from any earlier one. It lacks the cool precision of BYTNUM, but has a fiery eccentricity and some novel ideas that will appeal to some users, and comes closer to what I feel should be the ultimate goal of subroutine design: to return the maximum possible amount of information to the main program, whether or not this is needed by the program it was written to serve.

Sincerely,

H.T. Gordon
Agricultural Experiment Stn.
University of California
Berkeley, CA 94720

P.S. While I wish to make the software in this paper freely available without restriction to individual users, I retain full copyright for any commercial use. My motivation is not greed (my fee would be trivial and usually waived) but curiosity. Some time ago I sent a binary-to-decimal conversion program to MOS Technology, stating that it was to be in the public domain. Recently I asked them whether they had used it in their PET. The reply was that they did not plan to release any information on its programs. Puzzling, since anyone who buys a PET will be able to read its ROM. I am an admirer of their 6502 and KIM designs, and I sympathize with their desire to avoid hassles. Still, this gave me food for thought. Software is a trickier thing than the books and magazine articles for which copyright law was written, and in which changing a few words won't affect value. A seemingly minor software revision may so enhance value that its original will (and should) vanish forever. Like it or not, every programmer is part of a collective mind, and progress demands that he educate and be educated by others. There is no precise answer to the question: what is the value of a piece of software, and who owns what part of the value? In the software-cost controversy argued in *DDJ* by Tom Pittman and others, I strongly favor keeping all costs at the bare minimum and legal conflicts at zero, since I am primarily a user. Program costs are not easy to calculate. In my present "package", OPLEGL took a lot of time, BYTNUM much less because I was only modifuing Larry Fish's logic. NUMBYT was even easier because I knew what needed to be done, and SIMBUG was child's play. Still, all these things were simultaneously working themselves out in my mind, and without the basic (originally more grandiose) concept of a scanning-debugger I would never have bothered with OPLEGL. This concept was inspired by Jim Butterfield's screening-out of the 64 "easy" illegals in his relocator program. I was receptive to this because of earlier problems caused by an illegal opcode. So in a way SIMBUG is a costly program.

The real value of anything is not in its cost, but in its utility. It may cost a lot to produce a white elephant, but the result is still worthless. In our social system, utility tends to be measured in the marketplace, in dollars. This has worked well for hardware, and for any software that can be inextricably linked to hardware. Unattached software can be valued in dollars only to the extent that users can be compelled either to buy it or do without. But it is so readily diffusible and copiable that many users will not buy it except at a price little above the cost (in money and time) of copying. Copyrighted printed matter is now photocopied illegally by individuals with complete impunity, because enforcement is not practical. Where law fails, we cannot expect too much of ethics. So the utility (in the economic sense) of even the most useful "software" is low. One could say the same of rainfall and sunlight!

OPLEGL CORRECTION, AND A 6502 SCANNING-DEBUGGER

H.T. Gordon

The following corrected version of subroutine OPLEGL adds 4 bytes to screen out opcode 9E but saves 5 bytes by improved logic and structure, so is down to 66 bytes total. The task of saving the accumulator is left to the main program. The accumulator will return modified by from 2 to 6 successive logical-shift-rights. Otherwise, only the status register is affected. The Z flag is set only by A2 and the 5 legals of type X_1(4, A, C); these also set the C flag. The C flag is set by X91, 5, 9, D)⩾8D, by X(6,E)⩾A6, by $X_0$0⩾90, and by all X8, but cleared by all others. The N flag is always the complement of the C flag. The V flag is not affected. This status information may or may not be useful to the main program (who can tell?) but one ought to know that it exists.

Coding for OPLEGL

```
Ø27Ø 4A      OPLEGL  LSR A (bit Ø to C)
Ø271 9Ø Ø8           BCC TYPEØ2 (even #s)
Ø273 4A              LSR A (bit 1 to C)
Ø274 BØ 17           BCS ILLEGA (all
                          X(3,7,B,F))
Ø276 C9 22           CMP #$22 (LSRed $89)
Ø278 FØ 13           BEQ ILLEGA (89 illeg.)
Ø27A 6Ø              RTS (other X(1,5,9,D))
Ø27B 4A      TYPEØ2  LSR A (bit 1 to C)
Ø27C 9Ø 1Ø           BCC TYPEØ (X(Ø,4,8,C))
Ø27E 4A              LSR A (bit 2 to C)
Ø27F 9Ø Ø5           BCC TYPE2A (X(2,A))
Ø281 C9 13           CMP #$13 (LSRed $9E))
Ø283 FØ Ø8           BEQ ILLEGA (9E illeg.)
Ø285 6Ø      LEGALA  RTS  (other X(6,E))
Ø286 4A      TYPE2A  LSR A (bit 3 to C)
Ø287 BØ 2Ø           BCS TYP4AC (all XA))
Ø289 C9 ØA           CMP #$ØA (LSRed $A2)
Ø28B FØ F8           BEQ LEGALA (A2 legal)
Ø28D ØØ      ILLEGA  BRK (other X2 illeg.)
Ø28E 4A       TYPEØ  LSR A (bit 2 to C)
Ø28F BØ Ø8           BCS TYPE4C (X(4,C))
Ø291 4A              LSR A (bit 3 to C)
Ø292 BØ Ø4           BCS ALLOK (all X8)
Ø294 C9 Ø8           CMP #$Ø8 (LSRed $8Ø)
Ø296 FØ ØC           BEQ NOTLEG (8Ø illeg.)
Ø298 6Ø       ALLOK  RTS  (other XØ legal)
Ø299 4A      TYPE4C  LSR A (bit 3 to C)
Ø29A FØ Ø8           BEQ NOTLEG ($Ø4, $ØC,
                          LSRed to ØØ)
Ø29C BØ Ø7           BCS TYPEC (other XC)
Ø29E 29 ØD           AND #$ØD (tests LSRed
                              $44, $64)
Ø2AØ C9 Ø4           CMP #$Ø4 (now = Ø4)
Ø2A2 DØ Ø5           BNE TYP AC (other X4)
Ø2A4 ØØ      NOTLEG  BRK  (44, 64 illeg.)
Ø2A5 C9 Ø9    TYPEC  CMP #$Ø9 (LSRed $9C)
Ø2A7 FØ FB           BEQ NOTLEG (9C illeg.)
Ø2A9 4A      TYP4AC  LSR A (bit 4 to C)
Ø2AA 9Ø Ø5           BCC LEGIT (Xo legals)

Ø2AC 4A              LSR A (bit 5 to C)
Ø2AD C9 Ø2           CMP #$Ø2 (LSRed X=9 or
                          B now = Ø2)
Ø2AF DØ F3           BNE NOTLEG (≠, illeg.)
Ø2B1 6Ø       LEGIT  RTS (X1 legals)
```

It is natural that 6502 programmers would be more concerned about legality than the 8080 workers (who have only a dozen illegal opcodes to worry about). In any program, a wrong-bit bug can arise by miskeying or accidental bit-change during transfer to or from storage. In an 8080 program, such an error is likely to change an opcode to a different (wrong but legal) one. With 105 illegals there is a higher probability that a legal opcode will be altered to an illegal; furthermore, if a branch or jump should be misdirected to an operand location (by miscalculation, miskeying, or bit-dropping), the operand may not be a legal opcode. Even if it is, I shall in a subsequent paragraph indicate a way of proving that it is wrong. The underlying principle of a scanning debugger is that the inherent structural rigidities in a program make possible the automatic detection of certain errors, without any execution of the program. Execution of a program bug can cause a lot of damage in RAM – even *creating* bugs where there were none before – so that even an imperfect pre-execution debugging scan may well be worthwhile. The "opcode-type-counting" aspect of my program SIMBUG could be used for *any* microprocessor. The "legality-testing" aspect is especially valuable for the 650X. The 650X control unit, that knows all about the operands required by opcodes, knows nothing of legality and will cheerfully execute 93 of the illegals and be stopped cold by the other 12 (all type X2 except 82, A2, C2, and E2, of which only A2 is legal but all 4 are executed). Either way the result can be unpleasant, sometimes a subtle and far-reaching bug. It was such a bug that first aroused my interest in the executable illegals.

As an introduction to SIMBUG, I shall try to make the concept more concrete by analysing a short program segment, written in a schematic way as

$$L_1\ O_1\ O_2\ K_1\ O_3\ J_1\ K_2\ O_4\ B_1\ O_5\ K_3\ O_6$$

"O" indicates an operand, "J" a 1-byte opcode, "K" a 2-byte, and "L" a 3-byte. "B" indicates a branch opcode, a subclass of 2-bytes that is easy to detect (all 8 branches are type X_10) and has a very high usage frequency (10% or more of all the opcodes in most programs). All 4 types are *counted* by SIMBUG.

The debugger is initialized to pick up L_1. It calls OPLEGL to test its legality. If legal, it calls BYTNUM to determine where the next opcode is. If there are no illegality errors (that cause a program BREAK) it skips from code to code until it eventually encounters an illegal. No branches or jumps are taken (although more elaborate programming would make this possible). Operation can be terminated at any desired point by deliberately replacing a legal opcode by an illegal.

Let us now examine the effect of an opcode error (however caused) that is an alteration of a correct code to an incorrect but legal one, "W". If W specifies the *same* number of bytes, the error is undetectable. If W specifies 1 or 2 bytes, the debugger will pick up either O_1 or O_2 as the next opcode. Only if O_1 is a legal 2-byte, or O_2 a legal 1-byte, will it pick up K_1 as the third opcode and be back "on track". Such chance "error compensation" will sometimes occur. Note, however, that one 3-byte code has been converted into two (a 1-byte plus a 2-byte). In the example, the counts of 1-bytes and 2-bytes would be too high by 1, and the count of 3-bytes too low by 1. The programmer is alerted to search for wrong 3-byte. One can extend this analysis to every possible W-type error and see that it is highly improbable (though not impossible) that chance compensation will restore the correct counts. The difficulties are obvious: the programmer must *know* the correct counts, and in a long program the search for the opcode error will be tedious. The latter can be minimized if scanning is done in "chunks" of 256 bytes. This could be easily implemented in SIMBUG but I have not bothered, because this debugging concept is not yet (and may never be!) an accepted one. Improvement can wait until the simple version has (or has not) proved itself useful.

The only operand errors that a somewhat more complex version of SIMBUG could detect are the branch offsets, and then only if the error caused branching to an operand. The debugger could calculate the location to which the offset directed the operation, and start a scan from that address in the hope that mistracking would not be compensated but cause it to strike an illegal. A much more positive approach would be for a first pass of the debugger to record the address of every presumptive opcode in its search area, and in a second pass calculate the branch locations and compare them with the list of opcode addresses. Failure to find a match would be a guarantee of error, most probably in the offset. The gain in debugging power might not be worth the much higher memory cost. Only about 10% of all operands will be branch offsets, and the chance of an error detection would be roughly 50-50. It seems to be a characteristic of all debuggers (including human ones) that effectiveness is subject to severely diminishing returns, while costs increase exponentially.

Coding for SIMBUG

```
Ø2B3 AØ ØØ       SIMBUG  LDY #Ø
Ø2B5 A2 Ø7               LDX #7
Ø2B7 94 Ø2       CLERIT  STY CNTLO,X
Ø2B9 CA                  DEX
Ø2BA 1Ø FB               BPL CLERIT
Ø2BC B1 ØØ       LOADIT  LDA (BASAD),Y
Ø2BE 2Ø 7Ø Ø2            JSR OPLEGL
Ø2C1 B1 ØØ               LDA (BASAD),Y
Ø2C3 2Ø 1Ø Ø2            JSR BYTNUM
Ø2C6 F6 Ø2        COUNT  INC CNTLO,X
Ø2C8 DØ Ø2               BNE TSTBRN
Ø2CA F6 Ø6               INC CNTHI,X
Ø2CC 29 1F       TSTBRN  AND #$1F
Ø2CE C9 1Ø               CMP #$1Ø
Ø2DØ DØ Ø6               BNE INCAD
Ø2D2 E6 Ø2               INC CNTLO
Ø2D4 DØ Ø2               BNE INCAD
Ø2D6 E6 Ø6               INC CNTHI
Ø2D8 E6 ØØ        INCAD  INC BASAD
Ø2DA DØ Ø2               BNE NOPINC
Ø2DC E6 Ø1               INC BASAD+1
Ø2DE CA          NOPINC  DEX
Ø2DF DØ F7               BNE INCAD
Ø2E1 FØ D9               BEQ LOADIT

Zero-page locations affected:
ØØ BASAD     low scan-start address
Ø1 BASAD+1   hi   "    "      "
Ø2 CNTLO     low branch-opcode count
Ø3 CNTLO+1    "  1-byte-   "     "
Ø4 CNTLO+2    "  2-   "    "     "
Ø5 CNTLO+3    "  3-   "    "     "
Ø6 CNTHI     hi  branch-opcode   "
Ø7 CNTHI+1    "  1-byte-   "     "
Ø8 CNTHI+2    "  2-byte-   "     "
Ø9 CNTHI+3    "  3-   "    "     "
```

As befits a moronic main program, both logic and handling of SIMBUG are straightforward. The user inserts an illegal code at the point he wishes to scan to stop, keys in the start address in the BASAD zero-page locations, and runs SIMBUG. Almost instantly there will be a BREAK to whatever program-interrupt routine the user has decided on (this is system-dependent). The user checks the current address in BASAD. If it is a program-opcode location, this contains an illegal. This is fixed and SIMBUG is started again (the address in BASAD is right). If the illegal was at an operand location, derailing occurred at an earlier point, probably by a wrong but legal opcode calling for the wrong number of operand bytes. There is also a (presumably faint) possibility that a correct opcode was allotted the wrong number of operand bytes. This is hunted down and fixed. To play safe, it is a good idea to rescan from the start address. Eventually the break will occur at the pre-set terminator illegal. At this point, a track of legals exists in the program, but possibly not the intended one. A user who has not yet had his fingers burned may decide to execute the program, without bothering to check the opcode-type counts. Those who have previously gambled and lost will compare the counts from their listing with the SIMBUG counts, remembering that the computer is always right. If they agree, SIMBUG can do no more. True gamblers can strip SIMBUG of its 25-byte counting logic (the CLERIT and COUNT segments) and cut it to a super-moronic 23 bytes. The total byte-count of this minimal operation, including the two subroutines, is then 124. A less drastic stripping would eliminate only the branch-testing segment TSTBRN, saving 12 bytes.

SIMBUG was tested on the KIM-1 monitor programs. In the section starting at 1800, it was stopped by the illegal 0F at 1871. This is a data word, not an instruction. The counts of opcode-types checked perfectly. Starting a new scan on the program opcode at 1873, SIMBUG stopped at the data word 6B at 1BFA. It had scanned so many bytes that I did not bother to check the type-counts. A scan from 1C00 was stopped by the data word 4B at 1FDF. The scan had ploughed through a string of data zeroes well beyond the end of actual programming, so the type-counts would not be correct (but what can one expect from a minuscule moron?). Needless to say, when turned on itself and its subroutines, SIMBUG found no flaws!

As what I hope will be my swan song in this software area, I submit a radically different alternative to BYTNUM. Subroutine NUMBYT is fully relocatable, equally byte-efficient, but a bit slower. It includes some tricks that could be useful in other contexts. Users of the 6800 and 650X know that both designs have direct transfers between stack (S), accumulator (A), and status register (P), implemented in a different way: S-A-P in the 6800 but A-S-P in the 650X. The TAP instruction of the 6800 sets all 4 testable flags in P to the pattern of the 4 low-order bits in A (because the P register is organized as xxxxSZOC), and the elaborate conditional-branch instructions allow sophisticated bit-analysis. However, although the pattern is easily stored in S for later re-use (by a PSH ACA), it has to pass through (and so destroy the content of) A in order to move into P.

The 650X needs 2 bytes (PHA, then PLP) to move the A bit-pattern to P, and the 4 testable flags then reflect the status of the 2 highest-order and 2 lowest-order bits (because P is organized as NVxxxxZC). However, the pattern can be stored in S (either by deferring the PLP until it is needed, or by a PLP and PHP for both immediate and later use), and moved to P without affecting A.

The price paid for these powerful flag-setting operations is that non-testable flags are also affected (e.g., the interrupt-inhibit flag). In the 6800, the stack must be reset to the pre-call state (if the trick is used inside a subroutine) so that the RTS can pick up the correct return address. However, the 650X has the unique capability of transferring one byte, stored in the stack by a subroutine, to the main program. If an RTI is used instead of the usual RTS, the stack-stored byte is moved into P before the return address is picked up. The main program can use the status information immediately, and/or store it by a PHP for later use. NUMBYT uses these tricks both in its inner operation and to return much more information in its status register at exit than BYTNUM. Since it destroys the accumulator, the main program has the task of saving and restoring it if necessary. P at exit has 7 of the original bits in A, rotated to 0x765432, where x is always a zero bit and the original bit 1 is lost. The N, Z, and C flags therefore are = bits 0, 3, and 2.

PRAISE FOR JAMES & MICROTRONICS

Dear Jim: *Received: 77 Mar 29*

I'm glad to see you printing the letters from satisfied customers of companies as well as the ones from dissatisfied customers. Obviously it is as important for us to know the good outfits to go to as the bad ones to avoid.

Along those lines, I echo the sentiments of your other readers on James Electronics. If I order something on Monday, I have it by Friday or the following Monday.

Another good outfit is right there at Box 7454 in Menlo Park - - Microtronics. I had despaired of getting turnaround of anything less than a month on 1702A EPROM programming, after experiences with two other outfits, both in the Bay area. Microtronics gets them back in a week, and it only costs $3 (plus postage) for programming from a hex coding form! The best service at the lowest prices. They sell some hardware and software (on PROMS) as well as doing programming, but I haven't checked those products yet.

Jim Wilson 3250 Wing St. #402
Ketron, Inc. San Diego, CA. 92110

An optional modification of NUMBYT might be useful. Just before its fifth (PHA) instruction, bit 1 is in the C flag and bit 0 in the N flag. A 5-byte insertion: BCC SKIP, BPL SKIP, BRK will cause a program break for the 64 type X (3,7,B,F) illegals. This does not provide the complete "insurance" against illegality that OPLEGL gives, but costs much less in bytes and time. An added bonus is that if the N flag is set (bit 0 = 1), one knows the opcode was one of 64 "odds" of which only 1 (89) can be illegal, so that one could forego a full legality test at small risk. If the N flag is clear, the probability of an illegal is 20 times greater and a call to OPLEGL might be justifiable.

Coding for Subroutine NUMBYT

```
Ø24Ø A2 Ø1  NUMBYT LDX #1 (sets X reg.)
Ø242 18            CLC (clears carry)
Ø243 6A            ROR A (bit Ø to C)
Ø244 6A            ROR A (bit 1 to C)
Ø245 48            PHA (A to stack)
Ø246 6A            ROR A (bit 2 to C)
Ø247 6A            ROR A (bit 3 to C)
Ø248 BØ ØD         BCS NUMHAF (all
                       type X(8-F))
Ø24A 6A            ROR A (bit 4 to C)
Ø24B BØ 13         BCS 2BYTE (X1(Ø-7))
Ø24D C9 Ø4         CMP #4
Ø24F BØ ØF         BCS 2BYTE (all but
                       (6,4,2,Ø)Ø)
Ø251 C9 Ø1         CMP #1
Ø253 FØ ØA         BEQ 3BYTE  ($2Ø)
Ø255 4Ø            RTI ((6,4,Ø)Ø)
Ø256 28     NUMHAF PLP (stack to P)
Ø257 Ø8            PHP (P to stack)
Ø258 BØ Ø5         BCS 3BYTE (bit 2=1,
                       X(C,D,E,F))
Ø25A 1Ø Ø5         BPL 1BYTE (bit Ø=Ø,
                       X(8,A))
Ø25C 6A            ROR A (bit 4 to C)
Ø25D 9Ø Ø1         BCC 2BYTE (bit 4=Ø,
                       X.(9,B))
Ø25F E8      3BYTE INX
Ø26Ø E8      2BYTE INX
Ø261 4Ø      1BYTE RTI
```