52-NOTES

Volume 3 Number 2 48/39 February 1978

Newsletter of the SR-52 Users Club
published at
9459 Taylorsville Road
Dayton, OH 45424

PRACTICAL APPLICATION OF OBSCURE BUT POWERFUL PROGRAMMING FEATURES

In asking the question: Has anyone found a practical use for having a subroutine call itself? Cliff DeJong (290) suggests a broad topic for membership exploration: Given one of the more powerful but seldom-used machine capabilities, show how it can enhance the solution of a practical programming problem. Several somewhat obscure 58/59 capabilities come to mind: 1) Use of all 6 subroutine levels; 2) Use of both an indirect conditional and its indirect transfer together, i.e. Dsz*ab*cd; 3) Use of the list, trace, and Op 8 functions under program control; and 4) Use of HIR operations to alter stacked operands before pending arithmetic is executed. Members are invited to address these and to suggest other related topics covering any of the TI PPCs.

I'll lead off with some thoughts concerning Cliff's question. Given the architecture of the TI PPCs, I can't think of a practical application where having a subroutine call itself is an advantage. Sequences of the form: ...Lbl a ...b ...Lbl b ...rtn are indeed useful (see the V2N5p5 program), but amount to 2 routines sharing some common code, not one routine calling itself. A routine of the form: ...Lbl a ... a ...rtn calls itself, but without further specification, would not terminate. It could be made to terminate by writing something of the form: Lbl a seql ifflgx b stflgx a Lbl b seq2 rtn. Calling it with Flag x unset would cause the code designated by seql and seq2 to be executed as: seql seql seq2 seq2; calling with Flag x set would produce: seql seq2. More flags could be introduced to produce more variations, but I suspect the overhead of flag testing and setting would make straight forward calls to separate subroutines more attractive. Anyone else care to address Cliff's question?

EFFICIENT PRINT CODE STORAGE

Clyde Durbin (618) has found a HIR application which significantly minimizes data storage and processing requirements in many cases. His approach takes advantage of the differences in how Op 4 and HIR 08 treat display values during transfer to the print buffer. While Op 4 copies the 8 LSDs of the integer part of the display into the 8 LSDs of the print buffer (HIR 8), the HIR 08 function copies an entire real, as formatted in the display register. Thus only the print code contained in the integer part of the displayed mantissa is used to generate a character string with an Op 4, while as many as 8 LSDs (regardless of decimal point position) are used with HIR 08 storage. Using Op 4 to prepare one character string for printing, and HIR 08 on the same number for another not only saves the unpacking required in the V2N11p6 approach, but allows for overlapping that can enable one real to prepare 2 strings of up to 4 characters each. For example, Op 4 Op 6 on 916323117.4437 produces the tag DONE, while HIR 08 Op 6 on the same number produces NEXT. Note that the leading 9 serves as a dummy filler to force right justification within the 13 mantissa places, and that because of the odd number of mantissa places, overlapping with Op 5 printing would be difficult at best. However, there may be useful strings of characters some of whose print codes can be split and re-paired to form other useful strings. For example, 1624222437.253 produces DIGIT with Op 1 Op 5, but VGW÷π with HIR 05 Op 5 (sorry, I couldn't think of a more useful example!). Members are invited to share their best (most practical) Op 4/HIR 08 Op 6 and Op 1-4/ HIR 05-08 Op 5 creations.

RECORDING CURRENT MACHINE STATES (52,59)

In many practical applications it would be helpful to be able to record on mag cards such things as flag status; display format, angle mode, partitioning, printer connection, CROM module number, the contents of the HIRs and the T register ... prevailing conditions which the user might need to manually reinitialize each time he reads a particular card. Converting such information into recordable data is, for most of the above, a fairly straight forward exercise, but doing it efficiently may be somewhat challenging. For example, 10 registers could be designated to hold flag status: each stored with a 1 for set, 0 for not set, but this might be unacceptably wasteful of data registers. So it is attractive to consider data packing as an alternative. The following sequence stores the status of all 10 flags in Reg 1, taking advantage of the machine interpretation of flag 10 as flag 0 (TI-59): ... 0 S01 10 S00 1 S2 L1' CLR INV Ifflg*0 2' 1 L2' X R2 = SUM01 10 Prd2 Dsz0 1' ... and the following takes the value in Reg 1 and sets all 10 flags accordingly: ... CP 10 S0 L3' R1 ÷ 10 - Int S1 = x=t 4' Stflg*0 L4' Dsz0 3' ... . See V1N5p6 for a display-format-determining routine, V2N11p6 for angle mode indicator, and V2N9p2 and V2N10p3,4 for printer connection sensing. Partitioning can be decoded with: ... Op16 INV Int X 10 + 1 = Int S1 ,.. and set with:... R1 Op17 ... . Picking a suitable approach to saving the contents of the HIRs is somewhat frustrating: There ought to be a straight forward way to loop through a basic sequence 8 times, but there doesn't seem to be since the HIRs are not indirectly addressable under program control. The following sequence will transfer the contents of the HIRs to Reg 1-8 by an iterative process, requiring dynamic code modification (V1N2p3), but not efficiently: ,.. 10 Op17 10.82 S99 8 S0 L1' 10 Op17 1 SUM99 6 Op17 GTO 166 ... 168: S*0 Dsz0 1' ... . The straight forward: ... H11 S1 H12 S2 H13 S3 H14 S4 H15 S5 H16 S6 H17 S7 H18 S8 ... is both shorter and faster.

Members are invited to find and share better approaches, and to suggest other things worth recording; efficient recording utilities will be valuable to many users.

SORTING AND SEARCHING: SOME POPULAR APPLICATIONS (59)

While member feedback on the V2N8p1 and V2N11p2,3 articles has so far been slight, there is growing interest in developing efficient approaches to 2 related topics: 1) Addressing block-stored data by arbitrary tags, and 2) Sorting data by magnitude. Mechanization of both topics reveals machine as well as data dependency, and since practical implementation concerns more than just a few data, the discussion which follows centers on the TI-59/PC-100a.

Addressing Block-Stored Data by Arbitrary Tags: Bob Anderson (506) poses a common business problem, whose efficient solution should be of considerable interest to many users: A collection of entities, which I refer to here as records (machine-represented by specified numerical values), is to be stored in a specified block of data registers, each record identified by a unique tag or key (also machine-represented by a specified number) such that the required code and the execution time required to address each record from its key are minimized. The special case where keys match one for one the addresses of allocated data registers is simple enough to mechanize efficiently via indirect addressing, but it precludes the user from choosing "natural" keys: numbers and numerical representations of characters directly associated with the stored records. So let's look at a real-world situation which Bob poses: 20 departments in an organization are identified by the numbers (keys): 2,4,6,8,9,10,13,17,18,22,23,25, 29,31,32,33,34,51,72, and 73, each of which is associated with a labor rate (record). The records are to be stored in a concise block of data registers (not scattered throughout the Reg 2-Reg 73 range), and be efficiently addressable by their keys. A straight forward approach would be to compare an input key with all stored keys, one at a time until it is matched, at which time processing is diverted to appropriately initialize a pointer. But this would be prohibitively costly in both code and execution time. Another approach might be to try something along the lines of Ordered Hashing (V2N8p1), which Bob experimented with, using the ML-15 random number generator to serve as the hash function. This is an improvement, but is still a bit slow, and requires handling quite a few "collisions" (2 or more keys produce the same hash (register) address)..., a problem not likely to be solved by trying to find a better hash function. But assuming that the keys are not to be changed, in some cases advantage can be taken of the pattern they form. In this example, all but 3 of the 20 keys can be matched one for one with a register in the 1-34 address range, which might be an acceptably concise block. Each of the remaining keys: 51,72,73 should then be matched with one of the 14 unassigned registers in the 1-34 address range, and use made of the remaining 11 for program scratch. But what's the best way to match the remaining keys? It is tempting to look for a special no-collision hash function, which might be mechanized along the lines of the following algorithm:

```
    If $K \geq 51$ set $h(K) \leftarrow (K \bmod M) + 1$
    Else set $h(K) \leftarrow K$
```

which will do the job with M=47, and in English says: If an input key (K) is greater than or equal to 51, evaluate a hash function h(K) by adding 1 to the remainder modulo M (the remainder in integer form resulting from the division of K by M) of K. If K is less than 51, use K itself as the hash function value. In either case, the value h(K) points to the desired register. The number 47 was found by trial and error to satisfy the relations h(51)≠h(72)≠h(73), all in the 1-34 range of unassigned registers. The routine: LA S1 xXt 50 xGEt 1' R1 ÷ 47 = INV Int X 47 fix 0 EE INV EE = S1 Op21 L1' rtn accepts one of the 20 keys in the display, and following a call to A, produces the corresponding h(K) in Reg 1. Incidently, the fix 0 EE INV EE is required for rounding, and although fix 0 D.MS works just as well, it takes about 4 times as long to execute. The fix 0 can be eliminated if this routine is always called with a fix 0 display.

But it turns out that for only 3 keys, a straight forward match-and-process approach is considerably faster, and not much longer: LA S1 xXt 50 xGEt 1' 51 x=t 2' 72 x=t 3' 26 S1 rtn L2' 5 S1 rtn L3' 27 S1 L1' rtn. This exercise illustrates an important programming precept: Don't let exotic schemes close your mind to simpler ones that are better (unless your objective is to make an exotic scheme work!). One more thing to keep in mind when approaching the problem of how best to process arbitrary keys: It may pay off to spend a fair amount of effort looking for patterns which can be matched by simple non- or low-collision hash functions. There is a straight forward mathematical procedure for fitting n-1 data points with an nth degree polynomial, and while such an approach is impractical for large n, it suggests a compromise: Try out various combinations of simple mathematical functions: trig, log, arithmetic, etc; with perseverance and a little luck you may be able to solve the collision problem.

Sorting Data By Magnitude: In the previous problem, the user doesn't care where each record is stored, so long as it is readily accessible by an arbitrary key. Sorting data by magnitude is a different kind of problem, and arises any time the user wishes to start with an unordered sequential set of records and put them into ascending (or descending) order. While these records could also have arbitrary keys, such are not necessary to illustrate the sorting techniques discussed here, and in the interests of simplicity a record's key will be considered identical with the address of the register in which it resides. There are a few data dependencies to keep in mind when choosing an approach: 1) The initial ordering, 2) The number of repeated values, and 3) The number of data. These combined with machine idiosynchrasies make the choice of approach consequential. With no packing, about 100 data is a practical limit for the 59 to process, and for quantities in this neighborhood a variant of the so-called Shell Sort method does reasonably well. The program that follows was mechanized from an algorithm appearing in POPULAR COMPUTING (Jan 78 p5). Members wishing to examine Donald Shell's method in detail will find a technical discussion in Vol III pp 84-95 of Knuth's "The Art of Computer Programming". Use is made of the HIR operations (V2N9) to maximize data capacity, and as you may have noticed earlier in this issue, the mnemonic HIR has been shortened to H. A reverse-order input of 99 records takes about 21½ minutes to sort; a sample of 99 random numbers produced by ML-15 took 26½ minutes.

TI-59/PC-100A Program: Shell Sorting of up to 99 Data Ed

User Instructions: Key number of data (n), press E; key records (r_i), press R/S, for i=1,2,... n. Execution begins following the input of r_n, and when the program halts, Reg 1 - Reg n contain the input records sorted low to high, the contents of which may be displayed by successive R/Ss if a printer is not connectted, or are automatically printed if it is.

PROGRAM LISTING:

```
000:  LE xXt 10 Op17 CMs xXt H8 H7 S0 R/S S*0 Dsz0 015 H17 ÷ 2 = Int
028:  H7 CP x=t 104 1 H6 H18 - H17 = H5 H16 H4 H14 S0 R*0 xXt H17 +
059:  H14 = S0 R*0 xGEt 090 xXt S*0 H14 S0 xXt S*0 H17 H54 1 xXt H14
087:  xGEt 049 1 H36 H16 xXt H15 xGEt 045 GTO 022 1 INV List S0 R*0
111:  R/S Op20 GTO 109
```

Sorting permutations of a fixed quantity of specified values by magnitude is a special case of this second type, which Lou Cargile (625) has been exploring using several of the TI/HP PPCs to effectively perform card shuffling, dealing, and arranging for the game of Contract Bridge. His 59/100A program which follows combines a number of illustrative approaches and techniques, and will be a tough challenge to anyone aspiring to write a better one.

TI-59/PC-100A Program: Bridge Deal Lou Cargile (625)

User Instructions: Key RN seed LT 1, press E; (optional): key another seed LT 1, press D. To print one deal, press A; to print n deals, key n, press B.

Program Listing:

```
000:  LE' R*20 Op0 Op1 Op5 Op0 Op5 1 INV SUM20 rtn LA R19 SBR245 39
028:  S17 1 SUM25 R25 Prt CLR INV Stflg2 R19 X R26 + R27 = INV Int
051:  S19 + R16 = INV Int X 52 = Int S11 ÷ 13 = Int S12 + 1 = S0 R11
080:  + 1 - R12 X 13 = S13 R*0 ÷ R13 INV log EE S13 INV EE = INV Int
106:  X R13 = EE xXt CLR R13 ÷ 10 = xXt INV xGEt 036 xXt INV SUM*0
128:  xXt 1 INV SUM17 4 SUM0 xXt SUM*0 R17 ÷ 13 = INV Int xXt 0 INV
150:  x=t 036 xXt Stflg2 8 S0 E' 0 S24 4 S9 R*0 x=t 211 log EE Int S15
176:  INV EE INV log EE INV SUM*0 CLR R9 + 61 = S21 R15 + 47 = S23
201:  R*21 + R*23 X 100 = Op*9 SUM24 1 INV SUM0 Dsz9 166 R24 x=t 236
228:  Op05 4 SUM0 GTO 160 Adv Ifflg2 333 Dsz"22" A R/S S19 1 EE 13 INV
252:  EE ÷ 9 = S1 S2 S3 S04 3 S47 4 S48 5 S49 6 S50 7 S51 10 S52 11
285:  S53 12 S54 2.01 S55 25 S56 34 S57 26 S58 13 S59 9821 S26 .211327
326:  S27 69 S20 rtn R17 INV x=t 036 INV Stflg2 E' GTO 160 LB S22 GTO A
352:  LD S16 16 Op4 R16 GTO 395 LE xXt 6 Op17 CMs 7 Op17 Op0 R60 Op2
381:  R61 Op3 Op05 17 Op4 Adv xXt S19 Op6 Adv R/S
```

Pre-stored Data:

```
60:  143524 1622170000 36000000 23000000 16000000 15000000 4317363700
67:  3632413723 1713363700 3132353723
```

- - - - - - - - - - -

TIPS AND MISCELLANY

A Safe, Handy Hardware Interrupt (58/59): In cases where no CROM program is called within a loop, George Hartwig (638) has found a method paralleling the SR-52 D-R switch application (V3N1p4): A dummy call to a CROM program placed at a safe position in the loop works nicely with a manual RST to cause a transfer to step 000 of user memory.

For example, write: 000: R/S LA 20 S0 Ll' Dsz0 1' Pgm 7 SBR005 GTO A. Press A, and after an arbitrary time interval, press and hold down RST until execution halts. No matter when the RST is pressed, the halt occurs at the step 000 R/S, following a completed Dsz cycle. It appears that it doesn't matter at what step the CROM program is called; so long as the RST is depressed during the call, no CROM code is executed. If the CROM call is to an undefined label, an error condition is set, but transfer to step 000 of user memory still occurs. But for practical application, it is best to call a CROM program at a rtn (as in the above example), since this minimizes execution time (about 200 ms) and doesn't let the CROM code do anything during times when the RST is not depressed.

Use of p41 to Make Lbl Lbl Tricks Work for the 58/59: Rusty Wright (581) discovered that the SR-52 LBL LBL tricks can be made to work on the new machines if each Lbl Lbl sequence is preceded by the SST pseudo (p41). For example, Jared's flag reversal routine (V2N10p2) for the 52 can be written: ...Ifflg0 p41 Lbl Lbl INV Stflg0... for the new machines. Carl Paulson (854) has been exploring other sequences with p41 executed under program control, and finds that the first R/S or any number of p31s following one or more p41s is ignored. All this added to the unorthodox manual use of SST in creating fractured digits (V3N1p6) should help to encourage further exploration for new SST/p41 uses.

PC-100 to PC-100A: Mike Brown (128) reports that he turned his PC-100 into a PC-100A by positioning the 52-56 switch halfway between the 2 intended settings, effectively creating the "other" position on the PC-100A. The only problem Mike encountered was that sometimes the machine "... has trouble printing a line of 20 of the same characters", which he suggests may be due to a less robust power supply for the PC-100.

56/57/58 Program Exchange: Dave Johnston (5) has a new catalog dated 1 Feb 1978 listing 26 math, 8 statistics, 9 operations research and simulation, 24 physics, 10 other physical sciences, 24 engineering and technology, 3 life and behavioral sciences, 13 games, 6 finance, and 1 general info-test routine... programs written for the SR-56, of which 34 have been translated for the 57, and 15 for the 58. A few more programs in scattered categories were written for only the 57 or 58. Send Dave a SASE and 15¢ for the catalog. He continues to provide program copies at the modest rate of 5¢ per page.

CROM HIRs (V3N1p5): Several members have noted that T S Cox probably meant to refer to the Blackjack program (11) of the Leisure Library, and that while step 240 is a code 82, it is part of a GTO address, and was not intended to be a HIR operation. On the other hand, Roy Chardon's reported use of H8 in Pgm 13 of the Real Estate Library was intended (H8 at step 183 and H18 at step 219). In order to avoid confusion, members are asked when discussing CROM code to indicate whether a stated sequence is intended or unintended. Intended code is defined as that which the printer would list, addressed continuously from start to finish; unintended as printed starting from an intermediate step which separates the elements of a composite instruction.