52-NOTES

Volume 1 Number 5 48/48 October 1976

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

EDITORIAL

With four months and four newsletters behind us, I would like to re-examine Club purpose, goals, and operation.

All things considered, I see no reason to change the original purpose: to get more out of our machines by exchanging ideas. To date, more than 30 members have shared new ideas/provocative questions/useful information that have served as newsletter inputs in one way or another, and I am happy to take this opportunity on behalf of all the membership to thank them. Also, I appreciate the effort taken by many others in sending me material which for one reason or another has not been used. But how about the rest of you? You don't have to be an experienced programmer to discover something new... you mostly need some organized curiosity along with a fair amount of perseverance. I welcome and will publish whole programs that are not destined to join the TI PPX-52 library and which demonstrate, and are accompanied by descriptions of, new mechanizations. But I will continue to give priority to new, clever routines that have broad application. There is still much yet to be explored and discovered: useful pseudo sequences (see the article on pseudos elsewhere in this issue), optimum approaches to a host of combinatorial and numerical analysis problems, new computer programming teaching techniques, extended I/O via the printer interface terminal and card reader, and practical applications of programmed card read/write... just to name a few. I will continue to share my best ideas so long as I have reason to believe that a reasonable number of other members are doing the same. I don't want to see our Club become mostly a publisher-subscriber operation, and I hope most of you feel the same way. Most of what I've just said applies to the SR-56 too, only more so, since I don't expect to provide any input on this machine.

Monetary contributions at the rate of $1.00 per issue continue to provide an adequate operating fund, and I thank those who have contributed more, and who have been sufficiently pleased with newsletters to date to contribute toward publication past the first six issues. I do not intend to issue formal reminders for contribution renewal, but will assume that overdue members are no longer interested, and remove their names from membership and mailing lists. Incidently, I appreciate the fact that members living abroad have volunteered extra contributions when added postage is required.

Let me close this commentary by assuring all of you that I consider my time and effort to be well spent, and it is with pleasant anticipation that I look forward to getting inputs from members who have so far been among the silent majority, as well as to getting more gems from the productive minority.

The SR-52 Users Club is a non-profit loosely organized group of SR-52/56 owners/users who wish to get more out of their machines by exchanging ideas. Activity centers on a monthly newsletter, 52-NOTES edited and published by Richard C Vanderburgh in Dayton, Ohio. The SR-52 Users Club is neither sponsored nor officially sanctioned by Texas Instruments, Incorporated. Membership is open to any interested person, and a contribution of $6.00 brings the sender six issues of 52-NOTES.

TI's SR-52 Programming Workbook

TI is giving each PPX-52 member a Programming Workbook which it says normally sells for $4.95. It looks to me as though it might help the novice programmer get started, as it was designed to help SR-52 users who have had little or no prior programming experience. Basic programming concepts are explained in great detail and in very elementary terms. However, I would advise the reader/user to assume that while the sample routines and programs may demonstrate correct mechanizations, they are not necessarily examples of good programming. For instance, it goes without saying that the concept and use of subroutines are important aspects of programming. Yet, a poor example was chosen to illustrate the subroutine section of the workbook. The reader is left with the impression that a one-time calculation, the results of which are used twice, should be made into a subroutine and called twice.

An appendix of advanced programming techniques touches on some of the programming tricks and most of the previously unannounced features already familiar to many of us. Discussion is avowedly brief to discourage the novice from getting hopelessly entangled in exotic schemes. Unfortunately, some critical omissions are likely to cause, not suppress confusion. One such concerns code transferability, and is the omission of the requirement that an octet of program steps may not end in the code for a number if it is to be transferable (see V1N2p2). The workbook implication is that a 0, 2, 4, or 6 in the units place at the top of the octet is sufficient to make the octet transferable.

All things considered, I recommend this workbook to any SR-52 user who finds it difficult to understand the Owner's Manual and/or has done little or no programming, as an aid to getting started. But it is not likely to help anyone to write optimized/complex programs, and it might even be a hindrance.

PPX-52 Status

Word from TI indicates that the first catalog of SR-52 programs will be mailed October 25th to PPX members.

SHOOTING STARS SOLUTIONS

B H Andrews(20), Phil Sturmfels(49), Dwight Kucera(223), C A Matz(282), F W Lehan(300), and E S Molin(307) have all found that they can solve the Shooting Stars puzzle (V1N4p3) in eleven shots. Dwight suggests an analog to Shooting Stars which he refers to as the Big Bang in reverse: the starting "universe" is a black hole surrounded by stars; the end, a star surrounded by black holes. This appears to be an easier puzzle (no doubt helped by the un-forced first move), and Dwight has found a solution in five shots.

SHOOTING STARS PROGRAM ADDENDUM

Step 2 of the users instructions for the Shooting Stars program (V1N4p3) should include a "press A" statement that initiates processing.

SR-52 "CRASH"

Phil Sturmfels(49) and Chuck Sanford(214) have discovered that a sequence of the form: *LBL A *dsz A *LBL B can lead to an apparently unrecoverable "crash" (only the two minus signs show). Some reasonably large integer (99 or 500 or what have you) is first stored in Reg 00. Then the sequence A,HLT, B is keyed manually in RUN mode. If the display flashes, key CE and repeat A, HLT, B until the two minus signs are steadily displayed. The only known way out of this state is to turn the machine off.

PSEUDOS

A number of members seem to be having difficulty with pseudos, so I will attempt to pull together and expand previous explanations that were apparently not clear to all.

First, let me defend my choice of the word "pseudo" which in the context of SR-52 usage I define as any user-generated instruction code not directly keyable in LRN mode. One member (sorry, I don't recall who) suggests that the word "code" might be a better descriptor, since in computerese, pseudo refers to an instruction which is not directly executable, such as an assembler directive. Well, in this sense, most, if not all SR-52 instructions are pseudos since each starts a sequence of firmware instructions to obtain final results. However, the LRN mode keyable instructions, their associated machine instruction sequences, and execution results were presumably designed into the machine as predictable cause-effect entities and to me qualify under the heading of "SR-52 instructions", as contrasted with what I call"SR-52 pseudos" which set in motion unknown sequences of machine instructions, whose final results may only be partially evident to the user.

But let's press on to the problem of how to create pseudos. I suspect that members who have been unsuccessful so far have also experienced difficulty wading through the register behavior discussions (V1N1p4,p5 and V1N2p1,p2), so I'll avoid technical detail as much as I can, and try a step-by-step approach. If you have not already done so, I recommend writing down a table that shows which program steps are contained in which program memory registers: a reference that you can go to when you need to know that say, step 005 is in Reg 70, or that step 153 is in Reg 89, etc. In other words, write down the complete sequence implied by: 000-007 Reg 70, 008-015 Reg 71, ...216-223 Reg 97. Now, when you need a pseudo somewhere in a program you are keying in, first determine what program memory register it will be stored in. For instance, in the Streamlined Dynamic NIM game (v1N4p6), we find a requirement for a pseudo 84 at step 049. From our table, we see that step 049 is contained in Reg 76. The next thing to do is to create the pseudo as a run mode number which can be stored in the desired program-memory register. In our example, we need an 84 in Reg 76. By following the rules governing register behavior we could place the 84 directly at the desired step (second in our example) within the octet of steps comprising Reg 76 by keying: 1 EE 40 STO 76 8 EE 28 SUM 76. But it is probably easier, and in this case simpler just to key 84 STO 76 GTO 049, and then in LRN mode keying: *del *del *del *del *del *del. In either case any prior Reg 76 contents is destroyed, so the pseudo should be positioned before surrounding code is keyed in. For the NIM program, I recommend creating both pseudo 84s before writing other code. The second one is at step 174 (which is the seventh step in Reg 91), and only requires the sequence: 84 STO 91 GTO 174 LRN *del.

Pseudo <u>behavior</u> is something else, and so far as I know has only been superficially explored. Pseudo 84 appears to be a handy one-instruction error-condition producer. But does it do anything else? The sequence: *LBL A pseudo 84 HLT produces a flashing -1 -99 when executed, but an executed: *LBL A *pi X pseudo 84 HLT followed by a manually keyed: CE =, produces fractured digits (see V1N1p3 and V1N2p5). Now, this is just the beginning. What is the effect of preceding and following a pseudo 84 by all possible combinations of all the other "regular" instructions? Add to that all non-duplicating combinations of all 17 pseudos (V1N2p4) with the "regulars", and it is evident that it will be some time before we know all there is to know about pseudo behavior! (Ofcourse, there are probably quite a few interesting sequences of "regulars" that have yet to be discovered, as well)

PSEUDOS (con)

There may be an added complication concerning pseudos, as all SR-52s may not respond in the same way when either generating or executing pseudos. Anyone following the suggested steps above to write pseudo 84s in the NIM game who gets different or unworkable results should let me know, stating results obtained, and the machine serial number. Vince Salemme (206) reports difficulty with his machine (#14046) when attempting to fill Reg 70 and 71 with pseudo 26s by storing the number: -2.626262626 -62 in Reg 70 and 71. If you key the sequence: 2.626262626 +/- EE +/- 62 STO 70 STO 71, steps 000-015 should be: 26 06 00 26 26 26 26 26 26 06 00 26 26 26 26 26. Vince claims that step 000 shows up with pseudo 66 instead of 26. According to register behavior rules, a pseudo 66 is positioned directly at step 000 by storing a number like: -1 EE -6 in Reg 70. (The number must have both its mantissa and exponent negative and have a 6 in the units place of the exponent).

I invite members to experiment systematically with different sequences involving pseudos, and to send me results and conclusions. I'll attempt to put it all together in a future newsletter.

ADVANCED PROGRAMMING TECHNIQUES (Part II: Table Lookup Optimization)

The method used in the "Solution To 4 Simultaneous Equations" program (V1N4p2) to squeeze a lot of RCLs into a relatively small amount of program memory may be used to advantage in a broad range of applications. But, as in any attempt at memory-space optimization, it is important not to lose sight of overall objectives. If an efficient practical application is desired, there is no point in saving program steps when resulting execution time, I/O handling, or error accumulations are unacceptable. And it is even possible that the "overhead" required to control and run optimized routines makes the total number of program steps more than was required in the first place. On the other hand, if the primary objective is to demonstrate a programming technique, inefficiencies don't matter.

For the discussion that follows, let's assume that memory-space optimization is the primary objective. As the V1N4p2 program shows, it is not difficult to pack 6 register-pointers into one table-register. But how about more? If the radix (decimal) point is to be moved two places at a time to present desired pointers, 6 would appear to be the limit. In this case, each pointer can be in the range: 0-99. But suppose the entire range isn't required. In the V1N4p2 program, only registers 00 through 15 need to be pointed to. The capability of pointing to Reg 16 through 99 is unused, and is effectively wasted. Fortunately, this unused range can be traded for more pointers by using another number base. Moving the decimal point two places to the left is accomplished by dividing the number by one hundred. If instead, we move the radix by dividing by sixteen and perform a few base sixteen manipulations, we find it possible to pack up to ten pointers in the 0-15 range into one table-register. Unfortunately, we can't take advantage of the SR-52's ability to act only on the two least significant digits of the integer part of a large positive real (V1N3p1) when it indirectly addresses a register, since the two digits need to be in a base ten (or modified base 10^n, n=1,2,...10) system. Instead, we can effectively move the base sixteen radix point to the left in the decimal representation of a base sixteen number by successive divisions by sixteen, using the normalized remainder (the decimal remainder is multiplied by 16 and rounded up) each time as the next pointer, and

leaving a successively shrinking integer part for the next division. But in order to get 10 pointers per table-register this way, we need to carry 13 base ten places, which means using register arithmetic, and integer/fraction separation within a register. The "overhead" to do all this is significantly greater than that required if only ten places are carried, in which case we can only pack 8 pointers per table-register. So it becomes a trade-off. The following routines show what can be done, and what the overhead "ballpark" is: *LBL A + 16 *PROD 69 0 = SUM 69 *rtn *LBL B RCL 69 div 16 - INV *D.MS INV *D.MS *fix 0 *D.MS STO 69 = X 16 = *fix 0 *D.MS INV *fix *rtn *LBL C 16 INV *PROD 69 RCL 69 STO 68 STO 67 1 EE 12 SUM 67 INV SUM 67 RCL 67 STO 69 INV SUM 68 X 16 = INV EE *fix 0 *D.MS *rtn. Routine A packs pointers into Reg 69 and routines B and C unpack them: B for up to 8 pointers, C for 9 or 10. For example, to pack the pointers: 15, 3, 9, 13, 4, 8, 14, 1, 11, 7; key CLR, then each number followed by A. Reg 69 holds the integer: 1046315852215 which represents the ten pointers in packed form. These will only unpack correctly (in reverse order) if routine C is used (just press C for each pointer). Try a string of 8 pointers (in the 0-15 range) and unpack them with routine B. In a practical application, routine A would be used to help write the main program by generating the table-register constants to be permanently stored in program memory. These could either be manually stored as each constant is generated, transfering the contents of Reg 69 to the desired program-memory register, or routine A could be enhanced to do this automatically. Once the main program has been written, routine A is no longer needed, and therefore does not add to the overhead. As written, routines B and C would only unpack Reg 69. Whichever is to be used in a practical application needs another routine that steps through the permanent table-registers, passing via Reg 69 the required packed data to routine B or C for unpacking.

Let's assume that we have a main program in mind that requires a 100-element lookup table. We will design routine E such that each time it is called, it makes Reg 98 a pointer to a data register in accordance with a pre-determined sequence. We will also assume that the required lookup table has been stored in Reg 88-97 (assuming ten pointers per register). With the following initialization prior to its first call: 10 STO 00 88 STO 99 RCL 88 STO 69, routine E might look like: *LBL E C STO 98 *dsz *LBL *LBL *rtn 10 STO 00 1 SUM 99 *IND RCL 99 STO 69 *rtn. At each call (from a main program) routine E returns with Reg 98 pointing to the next desired data register. It is, ofcourse, up to the main program to indirectly address Reg 98 after each call to E and do what is desired with the retrieved datum.

Let's take the simple requirement that all the main program needs to do is sum the products of 50 successive pairs of 16 numbers in accordance with a predetermined ordering. If this were to be mechanized by a straight forward succession of 50 groups of steps of the form: RCL ab X RCL cd D, where *LBL D = SUM 19 *rtn is the called subroutine D and ab and cd assume values in the range 00-15, over 400 program steps would be required. But if we make use of routine E, then a main program A might look like: *LBL A *CMs 10 STO 00 88 STO 99 RCL 88 STO 69 *LBL *1' RCL 99 - 98 = *ifzro *LBL E *IND RCL 98 STO 18 E *IND RCL 98 X RCL 18 = SUM 19 GTO *1' which although not very efficiently written, gets the whole job done within available program memory. Incidently, the "*LBL E" sequence in routine A does no harm, provided the real subroutine E is nearer the top of program memory than routine A. If the body of routine E were placed following the *LBL E in routine A, a non-0 encountered by the *ifzro test would cause routine E to be executed twice (see V1N4p1).

TABLE LOOKUP OPTIMIZATION (con)

But would we have done better packing only 8 pointers per table-register, and having routine E call B instead of C? Well, we would save 21 steps using routine B, but would need 3 more table-registers, for a net loss of 3 steps: an acceptable alternative since there is enough program memory, and probably better since routine B runs faster than C.

What all this boils down to is that an optimized table lookup scheme should take into account at least three factors: 1) the number of table elements, 2) the range of values assigned to table elements, and 3) the availability of unused program memory that can provide room for speedier code. Thus if the V1N4p2 program were to be "optimized" with denser pointer packing, it would run slower, and there would be no gain, only a loss, as the program already fits on one card. But how about a 4 X 4 matrix inversion, or high-order polynomial curve-fitting programs? Table lookup optimization may be the only way to get them to fit on one card.

TIPS

Battery Charger Connecting: Brian Sullivan(247) reports erratic SR-52 behavior if the charger is plugged in after the calculator is turned on. This may be due to connector plug shorting as it is inserted into the socket. It is probably wise not to plug the charger in while the machine is on. Similarly, when using the printer, turn it on first with the calculator locked into place before turning the calculator on.

Exchanging Mag Cards: Members attempting to exchange mag cards recorded on different machines should be aware of at least two potential difficulties: 1) misreads may occur because of different drive/clocking rates, and 2) the fact that a number of SR-52s were manufactured with an implied multiplication following a closing parenthesis. This second problem can be mitigated if programs written utilizing implied multiplication are annotated accordingly.

Mag Card Static Charge: Phil Sturmfels(049) suggests that temporary buildups of static electricity on mag cards may account for their not reading or writing properly on occasion. On one occasion, Phil couldn't get a brand new card to be written on, so shot it with a Zero-Stat static-neutralizing device designed for phonograph records, and some minutes later succeeded in writing on it.

ROUTINES

Jared Weinberger(221) has come through again with another routine, although this time claiming modestly that it might win the "Trivia Award of the Year". I tend to disagree, since it performs a potentially useful service, and the mechanization provides an interesting insight to display manipulations. Jared calls his routine "Fixed Point Extractor", and it lists as follows: *LBL A (10 *LBL *1' (INV EE - 1) + +/- INV *log EE INV EE *ifzro *1' 0) *rtn. When run, this routine returns with an integer in the range 0-9 in the display which represents the display format at the time the routine was invoked. I would only add that the routine appears to work just as well without the inner parentheses. The outer ones are, ofcourse, to protect main-program pending operations if this routine is called as a subroutine.