Using TI FORTH to Solve a Unique Social Security Number Problem

...Lee Stewart

I ran across a problem about 25 years ago that seemed a worthy candidate for TI FORTH on the TI 99/4A computer. I planned to use the problem in a presentation of programming in TI FORTH to the Washington DC Area TI Users Group (later, Mid-Atlantic Ninety NinERS [MANNERS]). I had worked out a way to solve the problem, but failed to complete the project by the time of the presentation. I presented what I had and managed to drag a few folks into my thought processes and to explain some of the rudiments of TI FORTH—maybe even hooked one or two.

I had always intended to finish a few projects I had started so many years ago with TI FORTH and the TI 99/4A. One of these was to solve the problem alluded to above with TI FORTH and another was to produce a version of the *TI FORTH Instruction Manual* that would be easier to use and with known errors corrected. I recently restarted these projects after I discovered there was a pretty fair Internet presence of TI 99ers and in the process finished the problem, which is described below, and am well on my way to completing the *TI FORTH Instruction Manual* project—but I digress.

The problem was to find a Social Security Number (SSN) that is composed of all of the digits 1 – 9. The SSN was further required at each position, counting from the left with the leftmost position counting as position 1, to be evenly divisible by the position number. For example, were the SSN 123-45-6789, the number at the third position is 123 and should be evenly divisible by 3, which it is. This number fails to meet the division criterion at position four: 1234/4 leaves a remainder.

My initial thought was to write a program that would go through all the permutations of nine unique digits for a nine-digit number until a solution is found. This number of permutations is 9! = 362,880. The actual number of times the program would have to loop to arrive at the answer would likely be less than this maximum number. However, it occurred to me that there are several criteria derivable from the division criterion and the unique digits that will dramatically reduce the number of iterations required to hit on the solution:

1. The even-numbered positions must contain even digits. An odd number cannot be evenly divided by an even number.
2. There are 4 even-numbered positions and only 4 even digits; therefore, the odd-numbered positions must contain odd digits, the only remaining available digits.
3. Positions 1 and 2 never need to be tested. Any digit in those positions will meet the division criterion because of (1) and (2).

There are other derivative criteria; but, the above are what governed my programming. The maximum number of iterations with the above restrictions is now 5!4! = 2880, a 126-fold reduction! A further reduction in the maximum iterations would be realized by noting that the fifth position can only contain 5 because we are not using 0. That would be only 4!3! = 144 iterations; but, I had written the program that solved the problem before I realized this. Besides, the program is very fast as it is. As it turns out, without even considering the three restrictions above, the program takes just 2.3 times as long to solve the problem. In fact, one can continue deriving more restrictive criteria to make this a manually tractable problem; but, the point here was to write FORTH words to do the job.

After loading the FORTH screens following these comments, executing the word, **`FIND_SSN`** , finds the one and only SSN (381-65-4729) meeting the criteria in 7.5 seconds after trying only 222 permutations of digits! To prove this, run the last word, **`NEXT_SSN`** to continue searching for another SSN. It fails on the very next attempt.

Screen #1

```
0  ( Find SSN using 123456789 only once and divisible by position)
1  BASE->R DECIMAL : IT ;
2  ( digit storage array for growing number)
3  0 VARIABLE SSN 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 ,
4  ( digit available flag array)
5  0 VARIABLE DIGAVAIL 1 , 1 , 1 , 1 , 1 , 1 , 1 , 1 , 1 ,
6  1 VARIABLE CPOS ( current position in SSN)
7  0 VARIABLE STRT ( starting place for next digit in DIGAVAIL)
8  0 VARIABLE DONE  ( we're done: solution or failure)
9  ( SSN at CPOS as a double precision integer)
10 0 VARIABLE CURNUM 0 ,
11 0 VARIABLE SSN_ITER  ( keep track of iterations of NEXT_DIGIT)
12 : 2* DUP + ;    ( multiply by 2)
13 : 2DUP OVER OVER ; ( DUP 32-bit number)
14 : 2DROP DROP DROP ;
15 : 2! >R R ! R> 2+ ! ;                        -->
```

Screen #2

```
0  : 2@ >R R 2+ @ R> @ ;
1  : M+ 0 D+ ;  ( d1 n --- d2)
2  : MMOD M/MOD 2DROP ( d n1 --- n2) ;
3  : DS* ( d1 n --- d2)  ( assumes no overflow! OK here)
4       SWAP OVER * >R U* 0 R> D+ ;
5  : BLD_NXT ( n --- ) ( CURNUM * 10 + passed digit)
6       CURNUM 2@ 10 DS* ROT M+ CURNUM 2! ;
7  : BLD_CURNUM    ( build CURNUM from SSN)
8       0. CURNUM 2! ( zero CURNUM)
9       CPOS @ 1+ 1 DO  ( build CURNUM to CPOS)
10        I 2* SSN + @   ( current digit)
11        BLD_NXT LOOP ;
12 : BACK_UP     ( back up 1 level & rebuild CURNUM)
13      -1 CPOS +!  BLD_CURNUM ;
14
15                                                              -->
```

Screen #3

```
0  : ?SSN ( --- f ) ( check SSN so far)
1       CPOS @ 2 >
2       IF  ( only need to check if CPOS > 2)
3         CURNUM 2@ CPOS @ MMOD
4         IF 0 ELSE 1 ENDIF
5       ELSE
6  1
7       ENDIF ;
8  : PRT_ITER CR SSN_ITER @ . ." iterations" CR ;
9  : PRT_SSN  0 10 GOTOXY ."                "
10      0 10 GOTOXY CURNUM 2@ <# #S #> TYPE ;
11 : PRT_SOLN 0 9 GOTOXY ."                " ( clear "Working...")
12      10 10 GOTOXY ." = desired SSN." PRT_ITER ;
13 : ?SUCCESS CPOS @ 9 = IF 1 DONE ! 1  ( leave flag)
14      ELSE 1 CPOS +!  ( increment CPOS) 0  ( leave flag)
15      ENDIF ;                                 -->
```

Screen #4

```
 0 : XTRCT_CD   ( --- n )
 1       SSN CPOS @ 2* + DUP @ ( get current digit)
 2       0 ROT !   ( set SSN[CPOS] to zero) ;
 3 : INIT_STRT ( n --- ) -DUP
 4       IF  ( current digit non-zero?)
 5         DUP 2+ STRT ! ( STRT=current digit+2 to stay odd/even)
 6         2* DIGAVAIL + 1 SWAP ! ( put current digit back)
 7       ELSE   ( zero?)
 8         CPOS @ 2 MOD 2 SWAP -
 9         STRT ! ( init STRT=1 if odd, =2 if even)
10       ENDIF ;
11 : ?EXH ( n --- f ) STRT @ 9 > ;
12 : BKUP_TRY CPOS @ 1 =
13       IF  ( exhausted at level 1?)
14         1 DONE ! CR ." Failed to find a solution!" PRT_ITER
15       ELSE BACK_UP ( otherwise back up) ENDIF ;           -->
```

Screen #5

```
 0 : FIND_DIG 10 STRT @ DO   ( loop to find available digit)
 1         I 2* DIGAVAIL + DUP @
 2         IF  ( found a digit) 0 SWAP !  ( make it unavailable)
 3           I CPOS @ 2* SSN + ! LEAVE ( updt SSN & exit loop)
 4         ELSE DROP ( drop extra address) ENDIF
 5       2 +LOOP ;  ( stay odd/even)
 6 : ?HAVE_DIG ( --- n ) SSN CPOS @ 2* + @ ;
 7 : SSN_TRY ( n --- )  ( did we find a digit?)
 8     IF ( yes! Got a digit) BLD_CURNUM ( updt CURNUM /w fnd dgt)
 9        ?SSN IF PRT_SSN  ( print current, good SSN)
10           ?SUCCESS IF PRT_SOLN ENDIF ( print success)
11        ENDIF
12     ELSE    ( no digit this level)
13        BKUP_TRY   ( try to back up to previous level)
14     ENDIF ;
15                                                        -->
```

Screen #6

```
 0 : NEXT_DIGIT XTRCT_CD INIT_STRT ?EXH
 1       IF    ( exhausted this level?)
 2         BKUP_TRY
 3       ELSE
 4         FIND_DIG
 5         ?HAVE_DIG
 6         SSN_TRY
 7       ENDIF ;
 8 : FIND_SSN CLS 0 9 GOTOXY ." Working SSN:"
 9       BEGIN 1 SSN_ITER +! NEXT_DIGIT DONE @ UNTIL ;
10
11 : INIT_SSN ( initialize values to start from scratch)
12       0 DONE ! 1 CPOS ! SSN 20 ERASE 0 DIGAVAIL !
13       DIGAVAIL 20 + DIGAVAIL 2+ DO 1 I ! LOOP 0 SSN_ITER ! ;
14 : NEXT_SSN      ( look for additional SSNs)
15       0 DONE ! 0 SSN_ITER ! FIND_SSN ;              R->BASE
```