EduCALC

TN 51C

(714) 582-2637

27953 CABOT ROAD LAGUNA NIGUEL, CA 92677

HP 48 LOOP STRUCTURES - BY EXAMPLE

Often a program has to perform a task repeatedly, over and over. Examples are placing 1 to 100 on the stack, calculating interest from month to month, or adding numbers in a list. The program sequence that is repeated is called a loop, or more specifically, a loop structure. The HP 48 Owner's Manual describes a loop structure this way: "Loop structures are built with commands - called structure words - that work only when used in proper combination with each other." The HP 48 uses two types of loop structures, definite and indefinite. A definite loop is executed a fixed number of times, an indefinite loop tests to exit rather than exit based on a counter value. The loop-clause is executed if the test is non-zero. The HP 48 has two definite loop structures - each with two variations - and two indefinite loop structures. These are listed in table 1.

TABLE 1 - HP 48 Loop Structures

Definite Loops		Indefinite Loops	
START...NEXT__	P 1d	DO...UNTIL...END____	P 8a
START...NEXT__	P 5a		
FOR...NEXT____	P 6a	WHILE...REPEAT...END_	P 9d
FOR...STEP____	P 7b		

PART I - DEFINITE LOOP STRUCTURES

A basic part of the definite loop structure is the loop counter. This counter may increment by one or it may increment or decrement by a specified amount. The indefinite loop counter structure words are:

NEXT - increments loop counter by one for each loop-clause execution.
STEP - increments or decrements loop counter by specified amount for each loop-clause execution.

The first definite loop structure in TABLE 1 is START. START requires two inputs; the start and finish values of the loop counter. In a program the START...NEXT structure looks like this.

```
<< . . . start finish START loop-clause NEXT . . . >>
```

Where: start is the initital value of loop counter;
finish is the final, test value for loop counter;
loop-clause is the loop code-sequence to be repeated.

To key in the following examples find the loop structure words in PRG, BRCH in "S" and "G" series. Then NXT on "S" series or START, FOR, DO, or WHILE on "G" machines.

EXAMPLES OF START...NEXT.

1. MAKE FOUR COPIES OF 444 ON THE STACK.

   The program puts 444 on level one and execute the loop-clause DUP three times.

```
'SN1'     << 444 1 3 START DUP NEXT >>
             33 Bytes  # 8380h  0.02_s
```

   In this example the number, 444, is first placed on level one of the stack. The loop counter, a variable stored in the machine that you cannot see or use, starts with an initial value of one. The finish value of three is also stored. The start and finish values are "consumed" by the machine when they are stored by START. At this time the stack contains 444, 1, and 3 and you will see this if you single step the program. The loop-clause is simply DUP. NEXT compares the loop counter value with the finish value three. The first time through the program the result of the comparison (Fn1) is non-zero and the loop-clause is repeated a second time. The test is made again and the loop-clause is executed a third time. This time the loop counter is equal to the finish value 3 (the difference between the counter value and the finish value is zero) and program execution continues following NEXT. In this example it is the end of the program. The loop-clause is always executed at least once.

   Some times you may want a loop to be executed "forever." The definite loop structure may be executed "forever" by using a very large number as the finish value. The next example shows another way to loop "forever."

2. "FILL" THE STACK WITH THE NUMBER 444.

   'SN2' is a modification of 'SN1'. Replace the finish value 3 with a large number. This is done using a function called MAXR. MAXR will either show as 9.99999999999E499 or 'MAXR' depending on the setting of system flag -2. If clear, MAXR is a real number. All loop structures treat MAXR the same regardless of the setting of system flag -2. This is not true for all commands. The command SUB, for example, cannot use MAXR as an input unless flag -2 is set or MAXR is followed by ->NUM.

   The number 444 comes from the program. We could leave this out of the program and assume that the object on level one is the one to use.

Foot note:
(Fn1) A comparison is done by comparing levels one and two. The comparison or test takes the values from the stack (consumes them) and leaves a zero, or one (non-zero) on level one. The test is "done" by the NEXT command. Commands that perform tests are: SAME, =, ==, <, <, >, >, not equal.

```
'SN2'  << 444 1 MAXR START DUP NEXT >>
          33 Bytes # 5606h   10 seconds = 2,201 (on SX).
```

The 2,201 444's are placed on the stack by allowing the machine to run for 10 seconds. As the stack "fills up" the program will slow down and eventually you will run out of memory. With 56.1 K of memory the out-of-memory message halted the machine in five minutes 25 seconds loading 22,402 copies of 444 on the stack. This is a good exercise to do at least once to become familiar with the reaction of the operating system to running out of memory.

3. BUILD A TEXT LINE OF n SPACES.

```
'SN3'     << "" 1 ROT START " " + NEXT >>
            33.5 Bytes # C8DFh  16 spaces = 0.259_s.
```

The input, n, is used as the finish value with the program supplying the start value. The loop-clause is: " " +. This method is not the optimum solution to the problem but is suitable for short strings. Looping structures are usually the slowest solution.

4. PLACE 1 TO 4 ON THE STACK.

```
'SN4'     << 1 DUP 3 START DUP 1 + NEXT >>
             30 Bytes  # 3DA4h   0.028_s.
```

5. PLACE 1 TO n ON THE STACK.

```
'SN5'     << 1 - 1 DUP 3 ROLL START DUP 1 + NEXT >>
             37.5 BYTES  # 6B4CH   n = 4 in 0.034_s.
```

6. PLACE THE ODD NUMBERS 1 TO n ON THE STACK.

```
'SN6'     << 1 - 1 DUP 3 ROLL START DUP 2 + NEXT >>
            37.5 Bytes  # 6B4Ch   0.034_s.
```

Remember the input is the number of times the loop-clause is executed. If the odd numbers 1 thru 8 are desired the input is 8 divided by 2. What if the input is 4, 4.4, or 4.9?

7. PLACE THE LETTERS DCBA IN A TEXT STRING.

This may be done several ways. Use the ASCII decimal numbers (A=65, B=66, etc.) and the CHR command to create the letters.

```
'SN7'  << "" 65 DUP DUP 68 START CHR 3 ROLL + SWAP 1 + DUP NEXT DROP2>>
          68.5 Bytes  # A719h   0.133_s.
```

This is an example of working too hard to get the job done. It is left as an exercise for the student to improve this program. You will see this example using other loop structures later.

8. SUM THE ELEMENTS OF A LIST.

```
'SN8'     << OBJ-> 2 SWAP START + NEXT >>
             25 Bytes  # 6632h  { 1 2 3 4 5 } = 15 in 0.04_s.
```

The object-> command explodes out the numbers onto the stack with the number of objects in the list on level one. To add five numbers requires the addition to be done four times. This is accomplished by "counting" from 2 to 5, the 5 coming from the OBJ-> command. This 'technique' saves arithmetic and stack manipulations. Could you replace the sequence (- 1 DUP 3 ROLL) with 2 ROT using this technique in 'SN5' AMD 'SN6'? What are the byte savings? What is the speed increase?

9. WHAT BATTERY CAPACITY REMAINS AFTER 30 DAYS IF THE BATTERY SELF DISCHARGES 1 % PER DAY?

If you start with 100%, 99% remains after the first day. At the end of each day the capacity is 0.99 times the capacity at the beginning of the day. Our program starts with 100% and multiplies by 0.99 30 times. The input is 30 on level one. What capacity remains after 100 days?

```
'SN9'  << 100 SWAP 1 SWAP START .99 * NEXT >>
          46 Bytes  # F465h  30 days is 73.97% in 0.16_s.
                           100 days is 36.60% in 0.55_s.
```

10. HOW MUCH WILL BE IN A SAVINGS ACCOUNT IF $300 IS DEPOSITED FOR 18 MONTHS AT 6% PER YEAR COMPOUNDED MONTHLY?

An interest rate of 6% per year is 0.5% per month. Each month the deposit value is increaed by a factor of 1.005. The program input is: deposit (300) ENTER number of periods (18).

```
'SN10' << 1 SWAP START 1.005 * NEXT >>
          33 Bytes  # FA3Dh   18 periods is $ 328.18 in 0.09_s.
                              36 periods is $ 359.00 in 0.19_s.
```

Suppose you deposited $10 each month after the initial $300 deposit. What would the account balance be after one month, after 12 months, after 18 months? 'SN10' is easily modified to add $10 AFTER the interest is calculated for each period. See 'SN11'. When using a financial calculator this would emulate the "END" mode.

11. WHAT IS AN 18 MONTH SAVINGS BALANCE IF $300 IS INITIAL DEPOSIT, $10 IS DEPOSITED MONTHLY, AND COMPOUNDING IS MONTHLY AT 0.5%?

```
'SN11' << 1 SWAP START 1.005 * 10 + NEXT >>
          46 Bytes  # FEE1h    1 month  is $ 311.50 in 0.01_s.
                              12 months is $ 441.86 in 0.09_s.
                              18 months is $ 516.04 in 0.14_s.
```

The START...NEXT loop is useful to double check TVM calculations made on financial calculators. The HP 48 is fast enough to do a month to month calculation by "brute force."

The next structure in TABLE 1 is START...STEP. In a program the START...STEP structure looks like this.

```
<< . . . start finish START loop-clause step STEP . . . >>
```

EXAMPLES OF START...STEP

All the examples above may be replaced with START ... 1 STEP. The number before STEP determines the increment (or decrement) of the loop counter. If the number is positive the loop-clause is executed again when the counter is less than or equal to the finish value. If the STEP number is negative the loop-clause is executed again when the counter is greater than or equal to finish.

The three inputs of START...STEP - start, finish, step - may be supplied by using a variable instead of a real number either from the stack or from the program.

1. COUNT FROM s TO f INCREMENTING BY i.

```
'SS1'  << -> s f i << s s f START DUP 1 DISP .3 WAIT i + i
         STEP DROP >> >>
         84 Bytes  # C8Fh
```

The use of local variables provides a great deal of flexibility. The program may alter the start, finish and increment values.

2. SAME AS 'SS1' EXCEPT SHOW SHIFTED DISPLAY OF NUMBER ONLY. (add 9 spaces)

```
'SS2'  << -> s f i << s s f START DUP "         " SWAP + CLLCD 4
         DISP 1 WAIT i + i STEP >> >>  (9 SPACES)
         96.5 Bytes  # E3A5
```

3. HOW MANY RANDOM NUMBERS ARE REQUIRED IF THEIR SUM IS EQUAL TO OR EXCEEDS 10?

```
'SS3'  << 0 0 10 START 1 + RAND STEP >>
          38 Bytes  # DE02h  17 in 0.137_s, 20 in 0.161_s.
```

RAND is used as the STEP value. The RAND command produces a random decimal number between 0 and 1. This is added to the start value - initially zero. The STEP value is added each time through the loop-clause until the total reaches or exceeds the finish value 10. The loop-clause, 1 +, keeps track of how many random numbers are required. Since the "average" random number is 0.5 the expected "average" number the program will produce is 20.

The START loop structure doesn't make the counter available to the programmer. The FOR structure, however, adds this feature.

FOR requires three inputs in addition to the loop-clause. Start, finish, and counter. In a program the FOR...NEXT structure looks like this:

```
<< . . . start finish FOR counter loop-clause NEXT . . . >>
```

The counter is a variable of your choosing and the object following FOR will be used as the counter. A class member used DEPTH as an command not realizing that he had forgotten the counter in the FOR structure. He was very confused trying to debug the program! It is important to understand the relationships between FOR, NEXT, STEP, etc.

EXAMPLES OF FOR...NEXT.

1. PLACE THE LETTERS A THRU Z IN A TEXT STRING. (see 'SN7')

```
'FN1'  << "" 65 90 FOR c c CHR + NEXT >>
          55 Bytes  # 4B2Bh  0.635_s.
```

The ASCII numbers 65 thru 90 correspond to A thru Z. The counter starts with 65 which is stored as a local variable c. The first c following FOR defines the counter. The loop-clause is c CHR +. The first time through the loop starts with 65, converts 65 to an "A" and adds it to the empty list on level one of the stack. The second time through c becomes 66 and the process repeats until c becomes 90. At this point the counter is equal to finish and the program continues after NEXT which is the end of the program. The "" in the beginning of the program is the empty list that is used to build the string character by character.

From a programming effieciency view point an A-Z text string is 31 bytes and "zero" seconds. Loop structures offer byte count savings when larger strings are required. Another advantage of the loop is that many different strings may be built by changing the input.

The FOR...NEXT loop is more convenient than the START...NEXT loop using local variables. See example 'SS1'

2. DO THE SAME AS 'FN1' EXCEPT REVERSE THE LETTERS.

```
'FN2'  << "" 65 90 FOR c c CHR SWAP + NEXT >>
          57.5 Bytes  # 65DEh   0.652_s.
```

This program "cheats" in the sense that the string is built by adding to the beginning instead of the end by adding a SWAP to 'FN1'. Is this the only way to solve this problem? What have you learned about structures that you may apply to this problem?

3. DO THE SAME AS 'FN2' BUILDING THE STRING FROM THE END.

```
'FN3'  <<
```

Is there any advantage to using 'FN2' Vs. 'FN3'?

4. DISPLAY A TABLE OF NUMBERS AND THEIR SQUARES (n: n^2).

Provide the beginning and end of the table as inputs.

```
'FN4'  << FOR x x SQ x ->TAG NEXT >>
          33.5 Bytes  # B897h   1,10 takes 0.235_s.
```

The ability of having a number on the stack when you need it using local variables is convenient, reduces stack operations, and avoids mental or written stack diagrams. The ->TAG command reauires that the two inputs be in a specific order. By squaring the number first and bringing the number back again keeps the program simpler. An alternative program, 'FN5', using DUP and SWAP provides a comparison.

5. WRITE AN ALTERNATIVE TO 'FN4' ABOVE.

```
'FN5'  << FOR x x DUP SQ SWAP ->TAG NEXT >>
          34 Bytes  # 82Dh   1,10 takes 0.233_s.
```

The next structure in TABLE 1 is the FOR...STEP. The FOR...STEP loop structure is similar to the START...NEXT structure discussed above. The STEP feature allows a negative (decrement) number to be used. In a program the FOR...STEP structure looks like this.

```
<< . . . start finish FOR counter loop-clause step STEP . . . >>
```

EXAMPLES OF FOR...STEP.

1. COUNT FROM s TO f INCREMENTING BY i. See 'SS1'

```
'FS1'  << -> s f i << s f FOR c c 1 DISP 1 WAIT i STEP >> >>
          68.5 Bytes  # 9CEBh
```

This is 7.5 bytes (9%) shorter than the same program using START ... STEP.

2. HOW MANY BITS ARE REQUIRED FOR DECIMAL n?

This problem comes up more frequently than you might expect. Usually LOGs are used. This program was written by Joseph Horn. He admits that he simply asked the question, "What happens if the step and the counter have the same value?" He recognized that it was counting in binary!

```
'FS2'  << 0 1 ROT FOR c 1 + c STEP >>
          36.5 Bytes  # 17BCh  1024 is 11 BITS in 0.102_s.
               2^32 = 4,294,967,296 is 33 BITS in 0.304_s.
```

3. ADD THE INPUT n AS A TAG TO 'FS2'.

```
'FS3'  << DUP 0 1 ROT FOR c 1 + c STEP SWAP ->TAG >>
          44 Bytes  # C2EDh    1024 is 11 BITS in 0.116_s.
                               2^32 is 33 BITS in 0.319_s.
```

PART II - INDEFINITE LOOP STRUCTURES

TABLE 1 lists two indefinite loop structures. The first is DO ... UNTIL ... END. In a program the DO...UNTIL...END indefinite loop structure looks like this:

<< . . . DO loop-clause UNTIL test-clause END . . . >>

This looping structure executes the loop-clause, evaluates the test-clause, and if the result is true (non-zero) continues the program after the END. The DO...UNTIL...END loop structure does not use a loop counter. The loop-clause is executed an indefinite number of times. The END is vital and often "forgotten." The HP 48 won't allow an Indefinite Loop Structure in a program without the END.

EXAMPLES OF DO...UNTIL...END.

1. COUNT TO 100 TO TEST THE SPEED OF THE MACHINE.

```
'DU1'  << 0 100 -> n << DO 1 + DUP UNTIL N > END >>
          57 Bytes  # 60BEh  1.13_s.
```

The loop-clause adds one to the initial zero on level one, then makes a copy for the next increment. The upper limit, 100, is stored in a local variable n and the loop continues until level one is equal to or greater than 100.

2. COUNT BY ADDING ONE CONTINUOUSLY PLACING THE NUMBERS ON THE STACK.

```
'DU2'     << 0 DO 1 + DUP UNTIL 0 END >>
             30 Bytes  # 355Eh  10 seconds = 1,932 counts.
```

Using zero as the test guarentees that the test will never be true. What if the initial zero in the program was replaced with a -1? Would the "counter" stop? What would be the count values?

3. DISPLAY KEYCODE OF PRESSED KEY.

```
'DU3'  << STD DO UNTIL KEY END >>
          22.5 Bytes  # 5B2Fh
```

Standard mode (STD) sets the display for the integers produced by the program. The DO loop executes KEY until a key is pressed. Key returns a zero to level one until a key is pressed. This keeps the DO loop going. When a key is pressed the key code (row/columnn) is placed on level one. The zero is "consumed." The non-zero test concludes the DO loop which ends the program. The Key Code remains in the display.

4. COUNT BY ADDING ONE UNTIL A KEY IS PRESSED.

```
'DU4'     << 0 DO 1 + UNTIL KEY END DROP >>
             30 Bytes  # 3EA1h  10 seconds = 1,821 counts.
```

Level one starts as zero, one is added until a key is pressed. The KEY command stops the program and places the key code (row/column) of the key pressed on level one. The drop at the end of the program removes the key code showing only the count. If you are fast enough at pressing the keys you may count only once.

This example DO loop may be used as a game. Run the program twice for each player. The objective is to mentally estimate the same time between key pressings to obtain the same count. Run the game by pressing the menu key twice. Once to start, the second time to stop. When both tries are on the stack subtract the two numbers. The player with the smallest difference wins.

5. PLAY AN AUDIO ALARM UNTIL A KEY IS PRESSED.

```
'DU5'  << DO 500 .5 BEEP 350 .5 BEEP UNTIL KEY END DROP >>
          69.5 Bytes  # 7C2Hh
```

Press any key to stop this alarm. Remember that the BEEPs are 0.5 seconds. You may have to hold down the pressed key at least that long to stop the loop. The DROP at the end removes the key code returned by KEY. See 'DU4' above.

6. DISPLAY KEYCODE OF PRESSED KEY INCLUDING SHIFTED COMBINATIONS.

```
'DU6'  << DO 0 WAIT UNTIL 1 END >>
          27.5 Bytes  # 9699h
```

Keys pressed	Code	Keys pressed	Code
SIN	41.1	alpha, SIN	41.4
<-, SIN	41.2	alpha, <-, SIN	41.5
->, SIN	41.3	alpha, ->, SIN	41.6

This is a variation of 'DU3' that uses a special feature of WAIT - argument 0. It shows all possible shifted variations of a key. The ON key is different. It terminates the DO loop and leaves the test value (0) on the stack.

The looping structures discussed thus far perform the "stop looping" test AFTER executing the loop-clause. The implication of this is that the loop-clause is ALWAYS executed at least once. The WHILE ... REPEAT ... END loop structure performs the loop test BEFORE the loop-clause is executed. In a program the WHILE...REPEAT...END structure looks like this:

```
<< . . . WHILE test-clause REPEAT loop-clause END . . . >>
```

The loop-clause is executed only if the test-clause is non-zero, otherwise, the program continues after the END. The test clause may be a 1. An example is 'WR1' below.

EXAMPLES OF WHILE...REPEAT...END.

1. COUNT "FOREVER" STARTING AT ZERO.

```
'WR1'    << 0 WHILE 1 REPEAT 1 + END >>
            32.5 Bytes  # 24B7h  1,883 counts in 10 seconds.
```

2. ADD SPACES TO FRONT OF STRING TO MAKE 22 CHARACTERS.

If the text string is 22 characters or more no spaces should be added. Here the test is made first to insure that spaces are not added - unless needed - by the loop-clause.

```
'WR2'    << WHILE DUP SIZE 22 < REPEAT " " SWAP + END >>
            51.5 Bytes  # D38Ah  1 character string = 0.36_s.
                                21 character string = 0.024_s.
                                22 character string = 0.007_s.
```

PART III - COMPARING LOOP STRUCTURES

How do the six looping structures compare? The six programs below count by one "forever". Ten second counts are made and the six loops are compared for the "best" in terms of minimum bytes and fastest execution.

The ten second counts were obtained using an HP 48G - because it is louder than an HP 48GX - program, 'time', that waits two seconds, beeps, waits 10 seconds and beeps again. With your finger on the menu key the program is started. Listen in a quiet room free of distractions for the beep. Press the menu key when the beep is heard and count to seven. Move your finger to the ON key while waiting. Keep your eyes closed and listen for the second beep and press ON. Repeat the measurement as many times as is needed for the greatest accuracy. Ten times is usually adaquate. The spread of measurements is very small, usually only a few counts per thousand. This technique has been used for many years for timing calculator functions - usually executing a command one thousand times.

```
START...NEXT              << 0 0 MAXR START 1 + NEXT >>
                              27.5 Bytes # 99A9h 10 seconds = 2,035 counts.

START...STEP              << 0 0 MAXR START 1 + 1 STEP >>
                              30 Bytes  # 3FE5h  10 seconds = 1,839 counts.

FOR...NEXT                << 0 0 MAXR FOR c 1 + NEXT >>
                              32 Bytes  # 4717h  10 seconds = 2,039 counts

FOR...STEP                << 0 0 MAXR FOR c 1 + 1 STEP >>
                              34.5 Bytes # E090h 10 seconds = 1,840 counts.

DO...UNTIL...END      << 0 DO 1 + UNTIL 0 END >>
                              27.5 Bytes # 7186h 10 seconds = 2,104 counts.

WHILE ... REPEAT... END << 0 WHILE 1 REPEAT 1 + END >>
                              32.5 Bytes # 24B7h 10 seconds = 1,883 counts.
```

```
'time'  << 2 WAIT 1100 .1 BEEP 10 WAIT 1100 .1 BEEP >>
           75 Bytes  # 84F7h
```

The frequency of 1100 hertz is quite loud. Your delay in responding to the beep is cancelled by having start and stop beeps.

The optimum count-by-one loop structure is the DO...UNTIL...END. It has the highest count and and the lowest byte count. Second is START...NEXT. The slower START loop is probably due to the internal counter incrementing. Setting the test value MAXR to 9.9999999999E499 also adds a little time.

Jim Donnelley in "The HP 48 Handbook", 2nd Edition compares looping structures: "START loops save memory and execute faster than FOR loops for applications where access to the index is not needed" and "Loops ending with NEXT execute faster than those ending with STEP, because the increment value is always 1."

PART IV CONCLUSION

Becomming familar with the HP 48 loop structures requires study and practice. By working through these examples and comparing the different loop structures you will be able to make the optimum choice for your programs. Program byte counts and execution speeds are provided for each of the 28 examples.

The FOR...NEXT is probably the most popular because the loop counter is available to the programmer. START...NEXT is useful for many tasks, and there are eleven examples for this structure. The indefinite loop structures DO...UNTIL...END and WHILE ... REPEAT ... END add a test clause to permit controlled exiting.

The examples above are a mixture of demonstrations and practical programs. The more useful programs are: SN3, SN8, SN10; SS2; FN1, FN3, FN4; FS2; DU4; and WR2.

TABLE 2 - Loop Structure Syntax

START...NEXT - 3 inputs required.

```
<< . . . start finish START loop-clause NEXT . . . >>
```

START...STEP - 4 inputs required.

```
<< . . . start finish START loop-clause step STEP . . . >>
```

FOR...NEXT - 3 inputs required.

```
<< . . . start finish FOR counter loop-clause NEXT . . . >>
```

FOR...STEP - 5 inputs required.

```
<< . . . start finish FOR counter loop-clause step STEP . . . >>
```

DO...UNTIL...END - 2 inputs required.

```
<< . . . DO loop-clause UNTIL test-clause END . . . >>
```

WHILE...REPEAT...END - 2 inputs required.

```
<< . . . WHILE test-clause REPEAT loop-clause END . . . >>
```

TABLE 3 - Program Summary

Name	Description	Bytes	Page
SN1	Make Four Copies of 444 on The Stack	33.0	2a
SN2	"Fill" Stack with 444's	33.0	2c
SN3	Build a Space Text String	33.5	3a
SN4	Place 1 thru 4 on the Stack	33.0	3b
SN5	Place 1 to n on the Stack	37.5	3b
SN6	Place Odd Numbers 1 to n on the Stack	37.5	3c
SN7	Place Letters DCBA in a Text String	68.5	3c
SN8	Sum The Elements in a List	25.0	3d
SN9	What is Ni Cad Battery Capacity After 30 Days?	46.0	4c
SN10	What is Savings Balance if $300 is compounded monthly at 6% oer Year for 18 Months?	33.0	4c
SN11	Same as SN10 except $10 is deposited monthly	46.0	4c
SS1	Count From s to f Incrementing by i	84.0	5b
SS2	Same as SS1 except Shift Display 9 spaces	96.5	5b
SS3	How Many Random Numbers are Required if their Sum is equal to or greater than ten?	38.0	5c
FN1	Make A thru Z Text String (See SN7)	55.0	6a
FN2	Same as FN1 except reversed	57.5	6c
FN3	Same as FN2 Building the String From the End	?	dd
FN4	Display Table on n and n^2	33.5	6d
FN5	Alternate to FN4	34.0	7a
FS1	Count From s to f Incrementing by i (See SS1)	68.5	7b
FS2	How Many BITS Are Required For Decimal n?	36.5	7c
FS3	Add Input n as a TAG to FS2	44.0	7c
DU1	Count to 100	57.0	8b
DU2	Count to n By Adding and Leaving Numbers On The Stack.	30.0	8b
DU3	Display Keycode of Pressed Key	22.5	8c
DU4	Sound An Alarm Until A Key is Pressed	30.0	8d
DU5	Sound An Audio Alarm Until A Key is Pressed	69.5	9a
DU6	Display Keycode of Pressed Key Including Shifted Combinations	27.5	9b
WR1	Count "Forever" Starting with Zero	32.5	9d
WR2	Add Leading Spaces To String To Make String 22 Characters Long	51.5	10a
time	Ten Second Timer With Two Second Lead	75.0	10d

Excercises for the student:

1. Study the following examples and answer the questions.
   SN3, page 3a, Run out of memory; SN7, page 3c, rewrite program;
   SN8, page 3d, measure/evaluate SN5 and SN6;
   FN3, page 6d, rewrite FN2, space provided;
   FS2, page 7c, write a program that solves the problem another way.
   DU2, page 8b, test, answer questions.
2. Make up a real world, practical example for START...STEP.
3. Make up examples for START and FOR that use a negative STEP.