HP-71B Advanced Uses of STRNGLEX

Welcome to a new article, this time featuring ***STRNGLEX***, a most useful *LEX (Language Extension)* binary file for the ***HP-71B***, which provides much-needed string handling support to complement the sorely lacking set of built-in string functions. First, all 11 functions provided are summarily described for the sake of completeness, then the focus centers on 4 of its less known keywords, whose full syntax and capabilities are described with basic examples aplenty as well as more elaborate examples which are discussed in detail.

Introduction

STRNGLEX is a medium-size *(837 bytes) LEX (Language Extension)* binary file created and distributed by *Hewlett-Packard* for the ***HP-71B***, which greatly enhances the *HP-71B*'s string-handling capabilities. Once made available to the operating system (by copying it to RAM or being present in some plug-in ROM module), it does provide 11 string-related functions in the form of *BASIC* keywords immediately available to use either from the command line or in user's programs.

STRNGLEX includes the following 12 keywords (those ending in *$* return string values; else, numeric values):

#	Keyword	Syntax	Description
1	TRIM$	TRIM$(A$,[B$])	*Deletes all spaces (or the character specified in B$) from both ends of A$*
2	LTRIM$	LTRIM$(A$,[B$])	*Deletes all spaces (or the character specified in B$) from the left end of A$*
3	RTRIM$	RTRIM$(A$,[B$])	*Deletes all spaces (or the character specified in B$) from the right end of A$*
4	LWC$	LWC$(A$)	*Converts A$ to all lowercase characters*
5	LWRC$	LWRC$(A$)	*Converts A$ to all lowercase characters (same as LWC$, alternate spelling)*
6	MEMBER	MEMBER(A$,B$,[C$])	*Returns the first position in A$ in which a character of B$ does appear*
7	SPAN	SPAN(A$,B$,[C$])	*Returns the first position in A$ in which a character not in B$ does appear*
8	REV$	REV$(A$)	*Returns the characters in A$ reversed from left to right*
9	ROT$	ROT$(A$,N)	*Returns the characters in A$ rotated by N positions in either direction*
10	RPT$	RPT$(A$,N)	*Returns the characters in A$ repeated N times*
11	SBIT	SBIT(A$,M,N)	*Returns the specified bit of A$*
12	SBIT$	SBIT$(A$,M,[N,[P]])	*Sets, clears or toggles the specified bits of A$*

[] *means the parameter is optional; A$,B$,C$ can be any string vars/expressions; M,N,P can be any numeric vars/expressions*

Originally these extra string functions were intended to be included as part of the 64 Kb system ROM but they were ultimately removed to make room for the (allegedly useless) *"CALC Mode"* feature, which required no less than 5 Kb of ROM to implement, so complex-number handling had to be removed as well. Fortunately, *HP* provided full complex-number functionality (except for some glaring omissions such as inverse trigonometric functions of complex arguments) as part of the 32 Kb plug-in *Math* ROM, while the extra string functions were released as extensions to the built-in set in the form of the *STRNGLEX* binary file discussed here.

Seven of the functions (`[L|R]TRIM$`, `LW[R]C$`, `REV$`, `ROT$` and `RPT$`) are quite elementary (if very useful) and present no difficulties to understand and use so this article focuses on the remaining four functions, (`MEMBER`, `SPAN` , `SBIT` and `SBIT$`), which are rarely used in programs despite being extremely useful in the appropriate context, explaining in detail what they do and providing a wealth of basic examples of use as well as several non-trivial, advanced applications.

MEMBER

MEMBER(A$,B$) returns the first position in *A$* in which any character present in *B$* does appear.

Note: *A$, B$* can be any string variables or string expressions.

Basic Examples

`MEMBER("ARBOTANTE**J**O","**J**")`	returns	10	*(the position of the **J**)*
`MEMBER("ARBOTANT**E**JO","J**E**")`	returns	9	*(the position of the **E**)*
`MEMBER("AR**B**OTANTEJO","JE**B**")`	returns	3	*(the position of the **B**)*
`MEMBER("ARBOTANTEJO","C")`	returns	0	*(the **C** does not appear)*
`MEMBER("**A**RBOTANTEJO","C**A**")`	returns	1	*(the position of the first **A**)*
`MEMBER("ARBOTANTEJO","DC")`	returns	0	*(neither **D** nor **C** do appear)*
`MEMBER("AR**B**OTANTEJO","DC**B**")`	returns	3	*(the position of the **B**)*
`MEMBER("**A**RBOTANTEJO","DC**A**")`	returns	1	*(the position of the first **A**)*
`MEMBER("","A")`	returns	0	*(A$ has 0 length)*
`MEMBER("A","")`	returns	0	*(B$ has no characters)*
`MEMBER("","")`	returns	0	*(A$ has 0 length)*

Note: if *A$* is the *null* string *""* then it always returns ***0*** regardless of the contents of *B$*.

Usage Example

Given a prime number, we want to check if every permutation of its digits is a prime number as well. As generating and checking all permutations can be a somewhat involved and lengthy procedure we want to first quickly filter out those numbers which definitely have at least one permutation that can't possibly be a prime.

This will be the case if the original prime includes any of the digits 0,2,4,5,6 or 8 because if it does they'll eventually end up as the last digit in some permutation and thus the corresponding number will be divisible by 2 and/or 5 and so it can't possibly be a prime and should be discarded without further processing.

The following test code uses `MEMBER` to implement such a filter:

```
10 DESTROY ALL @ INPUT A$
20 IF MEMBER(A$,"024568") THEN DISP "No good, next !" ELSE DISP "OK"
30 GOTO 10
```

```
>RUN
? 97135773      END LINE → No good, next !    (includes a 5)
? 3755130 79    END LINE → No good, next !    (includes a 0)
? 9135512 97    END LINE → No good, next !    (includes a 2)
? 7133397117    END LINE → OK                 (there's no 0, 2, 4, 5, 6 or 8)
```

SPAN

SPAN(A$,B$) returns the first position in *A$* in which any character **not** present in *B$* does appear.

Note: *A$*, *B$* can be any string variables.

Basic Examples

`SPAN("A`**`R`**`BOTANTEJO","A")`	returns	2	*(the position of the **R**)*
`SPAN("`**`A`**`RBOTANTEJO","R")`	returns	1	*(the position of the first **A**)*
`SPAN("ARB`**`O`**`TANTEJO","ARB")`	returns	4	*(the position of the first **O**)*
`SPAN("ARBOTANTE`**`J`**`O","ARBOTANTE")`	returns	10	*(the position of the **J**)*
`SPAN("ARBOTANTEJO","ARBOTANTEJ")`	returns	0	*(all appear in B$)*
`SPAN("","A")`	returns	0	*(A$ has 0 length)*
`SPAN("`**`A`**`","")`	returns	1	*(the position of the **A**)*
`SPAN("","")`	returns	0	*(A$ has 0 length)*

Note: if *A$* is the *null* string *""* then it always returns ***0*** regardless of the contents of *B$*.

Usage Example

Given two strings of the same length and each individually having no repeated characters, we want to quickly check if they are a *permutation* of each other.

Usually this would require a loop to check if every character of the first string does appear somewhere in the second string and *vice versa* but we can use `SPAN` to perform the check very quickly, with no loop required.

The following test code uses `SPAN` to implement such a filter:

```
10  DESTROY ALL @ INPUT A$,B$
20  IF SPAN(A$,B$) OR SPAN(B$,A$) THEN DISP "Not a permutation" ELSE DISP "OK"
30  GOTO 10
```

```
>RUN

? AMOR,ROMA              END LINE  →  OK                  (same chars, just rearranged)
? AMOR,AMAR              END LINE  →  Not a permutation   (the second string has no O)
? X,Y                    END LINE  →  Not a permutation   (the second string has no X)
? X,X                    END LINE  →  OK                  (both strings are equal)
? 81437652,74315826      END LINE  →  OK                  (same chars, just rearranged)
? 12345678,8765432O1     END LINE  →  Not a permutation   (the first string has no 0)
```

SBIT and SBIT$

SBIT(A$,I,N) returns the value *(0 or 1)* of the *N-th* bit (*0-7, 0=lowest bit*) of character *A$[I,I]*

SBIT$(A$,I) toggles all 8 bits of the *I-th* character of *A$*, i.e., of character *A$[I,I]*

SBIT$(A$,I,N) toggles the *N-th* bit (*0-7, 0=lowest bit*) of character *A$[I,I]*

SBIT$(A$,I,N,B) sets the *N-th* bit (*0-7, 0=lowest bit*) of character *A$[I,I]* to the value *B* (*0 or 1*)

Note: *A$* can be any string variable*; I,N,B* can be any numeric variables/expressions.

Example 1

The following sample code inputs a 2-character (or longer) string, displays a line consisting in an arbitrary line label *(-1)*, the given string, the numeric *ASCII* code for its 2nd character and its bit representation, then loops through each bit of that character in the original string from the lowest *(bit 0)* to the highest *(bit 7)*, setting it to ***1*** and displaying in turn each line (labeled with the bit number, from *0* to *7*) showing the result of the operation.

Note: the `BSTR$` keyword from the *Math* ROM is used in subprogram ***PR****(int bits)* below to simplify the generation of the output. Also, the `WHILE / END WHILE` keywords from the *JPC* ROM are used for simplicity as well.

```
10  DESTROY ALL @ INPUT A$ @ K=2 ! second character of A$
20  CALL PR(A$,K,-1) @ DISP @ FOR I=0 TO 7 @ B$=SBIT$(A$,K,I,1)
30  CALL PR(B$,K,I) @ NEXT I @ GOTO 10
40  !
50  SUB PR(A$,I,B) @ N=NUM(A$[I,I]) @ B$=BSTR$(N,2) @ WHILE LEN(B$)<8
60  B$="0"&B$ @ END WHILE @ DISP "  ";B;"  ";A$;"  ";N,B$
```

```
>RUN

        ? ABC    END LINE

       -1   ABC   66     01000010   (the original string, B is the 2nd char = ASCII 66 = 01000010)

        0   ACC   67     01000011   (bit 0 is set to 1 so the B changes to a C, ASCII 67)
        1   ABC   66     01000010   (bit 1 is set to 1, which it was so the B is left unchanged)
        2   AFC   70     01000110   (bit 2 is set to 1 so the B changes to a F, ASCII 70)
        3   AJC   74     01001010   (bit 3 is set to 1 so the B changes to a J, ASCII 74)
        4   ARC   82     01010010   (bit 4 is set to 1 so the B changes to a R, ASCII 82)
        5   AbC   98     01100010   (bit 5 is set to 1 so the B changes to a b, ASCII 98)
        6   ABC   66     01000010   (bit 6 is set to 1, which it was so the B is left unchanged)
        7   A.C   194    11000010   (bit 7 is set to 1 so the B changes to a ., ASCII 194)
```

Example 2

The SBIT keyword can be used as a replacement for the BIT keyword of the *HP-IL* ROM, as follows:

BIT(N,B)	*is equivalent to*	SBIT(CHR$(N),1,B)
SBIT(A$,I,B)	*is equivalent to*	BIT(NUM(A$[I]),B)

so you can do basic bit handling without having to use the *HP-IL* ROM. The following tester code checks that indeed BIT and SBIT are equivalent by generating all byte values from 0 to 255 and comparing the values returned by both BIT and SBIT for all 8 bits in each byte:

```
10  DESTROY ALL @ FOR N=0 TO 255 @ FOR B=0 TO 7
20  IF BIT(N,B)#SBIT(CHR$(N),1,B) THEN DISP "Not equivalent" @ END
30  NEXT B @ NEXT I @ DISP "Equivalence OK"
```

```
>RUN
        Equivalence OK
```

Example 3

We want to be able to find out as quickly as possible whether a given positive integer from 2 to some maximum value is prime or not. One way to do it using SBIT and SBIT$ is to generate a bit array stored in memory as a string, with each bit set if the corresponding index is a prime and clear otherwise. SBIT$ can be used to populate the bit array initially, and afterwards SBIT can be used to check the primality of any given integer in the range by simply finding out whether the corresponding bit is set or not. This has the following advantages:

- each number in range requires just *one bit* to indicate whether the number is a prime or not
- the primes themselves need not be stored, they're implicit as the bit array indexes
- as all even numbers >2 aren't prime we only need to store the prime status of the odd values
- the primality of any given number can be instantly retrieved with one lookup, no search or loops
- this technique will work not only for primes but generally for any sequence of integers, see **Notes**.

The following sample code implements the concept and checks the implementation. Usually the bit string recording the primality status would be already created and placed in memory but as this is *proof-of-concept* code we'll first create the bit string at the start (using the PRIM function of the *JPC* ROM to simplify the generation) and then we can check the implementation and find out the primality of any given values, like this:

This line creates a 1,000-character string and initializes all its bits to 0:

```
10  DESTROY ALL @ DIM A$[1000] @ A$=RPT$(CHR$(0),1000)
```

For every odd integer in [3, 16000], set the corresponding bit in the string if the integer is prime:

```
20  FOR N=3 TO 16000 STEP 2 @ IF PRIM(N) THEN 40 ELSE I=(N-1)/2
30  A$=SBIT$(A$,I DIV 8+1,MOD(I,8))
40  NEXT N
```

For a quick visual check, this displays just the primes up to 100 using the FNP function defined below:

```
 60  FOR N=2 TO 100 @ IF FNP(N) THEN DISP N;
 70  NEXT N @ DISP
```

To test the correctness of the implementation now we check for primality all integers in [2, 16000] using both the user-defined function FNP and the PRIM function, displaying any discrepancies (there should be none):

```
 90  FOR N=2 TO 16000 @ IF FNP(N)#NOT PRIM(N) THEN DISP "ERROR: Mismatch";N
100  NEXT N @ DISP "Implementation tested"
```

*Finally we time the implementation by counting the number of primes in [2, 16000] via the user-defined function FNP(N) which returns **1** (True) if N is prime and **0** (False) otherwise, displaying both the count and the timing:*

```
120  SETTIME 0 @ S=0 @ FOR N=2 TO 16000 @ S=S+FNP(N) @ NEXT N @ DISP S;TIME @ END
130  !
140  DEF FNP(N) @ IF N=2 THEN FNP=1 @ END ELSE IF NOT MOD(N,2) THEN FNP=0 @ END
150  N=(N-1) DIV 2 @ FNP=SBIT(A$,N DIV 8+1,MOD(N,8)) @ END DEF

>RUN

  2  3  5  7  11  13  17  19  23  29  31  37  41  43  47  53  59  61  67  71  73 79  83  89  97

  Implementation tested        (no mismatches were displayed, so the implementation is correct)

  1862  7.21        (1,862 primes counted in 7.21 sec. using J-F. Garnier's Emu71 HP-71B emulator)
```

Notes

1. As stated, this technique can be used for arbitrary increasing sequences of positive integers with no repeated elements, not just the primes. Once the bit array is created and in memory, any value can be checked almost instantly to see whether it belongs in the sequence, with no search at all, no matter how difficult or time-consuming it was to create the bit array initially.
2. This is much faster than storing the sequence elements in an array and having to loop through them to see if some value is in the array or not, either using a linear search (931 tries average for our 1,862 primes example) or even a binary search (~11 tries average). With this packing technique, just a single try is ever necessary.
3. The memory needed for a sequence of up to N elements is just N/8 bytes generally though some significant savings can be applied depending on the sequence, as in our example where only odd numbers have to be considered and we can reach up to N using just N/16 bytes of RAM. On the other hand, storing the primes themselves in an integer array would take 3 bytes per element (for values up to 99,999) so 1,862 primes would need 5,586 bytes of memory to store them, as compared with just 1,000 bytes with this technique.

Copyrights