A Portable Stack-based String Library for Forth Systems

Simplifying string handling through the use of a String Stack

Mark Wills

2/27/2014

String handling is not one of Forth strong points. Out-of-the-box support for strings is all but non-existent in standard Forth. Whilst the concept of strings does exist in the language, relatively few words are provided to allow effective string manipulation; the normal approach for Forth programmers is to roll one's own string functions as required. Issues such as heap allocation and de-allocation, and memory fragmentation are thorny issues which are often passed over in preference for a 'quick-and-dirty' solution that solves the problem at hand. Presented here is an ANS94 Forth compliant library which affords the Forth programmer such facilities as string constants, transient strings, and a wide range of string manipulation words. Issues such as memory allocation, memory de-allocation and memory fragmentation are rendered irrelevant through the provision of a string stack, which is used to host and manipulate transient strings. Specific attention has been paid to producing code that is portable, and compliant with the ANS94 specification, in order that it may be used with no, or minimal changes on a variety of Forth systems.

Table of Contents

Introduction – The Concepts behind the Library....4
Conventions....4
Stack Notation....4
Acknowledgements....4
String Constant Words....5
$CONST (max_len tib:"name" --) (runtime: -- $Caddr) “string constant”....5
MAXLEN$ ($Caddr -- max_len) “maximum length of string”....5
:=” ($Caddr tib:"string" --) “assign string constant”....5
.$CONST ($Caddr --) “display string constant”....5
CLEN$ ($Caddr – len) “string constant length”....6
>$ ($Caddr --) (ss: -- str) “to string stack”....6
String Stack Words....7
$" (tib:"string" --) (ss: -- str) “string to string stack”....7
DUP$ (--) (ss: s1 -- s1 s1) “duplicate string”....7
DROP$ (--) (ss: str --) “drop string”....7
SWAP$ (--) (ss: s1 s2 -- s2 s1) “swap string”....8
NIP$ (--) (ss: s1 s2 -- s2) “nip string”....8
OVER$ (--) (ss: s1 s2 – s1 s2 s1) “over string”....8
ROT$ (--) (ss: s3 s2 s1 -- s2 s1 s3) \ "rotate strings"....9
-rot$ (--) (ss: s3 s2 s1 -- s1 s3 s2) \ "rotate strings"....9
>$CONST ($Caddr --) (ss: str --) “to string constant”....9
+$ (--) (ss: str1 str2 – str2&str1) ”concatenate strings”....9
LEN$ (-- len) (ss: --) “length of string”....9
MID$ (start end --) (ss: str1 – str1 str2) “mid-string”....10
LEFT$ (len --) (ss: str1 – str2) “left of string”....10
RIGHT$ (len --) (ss: str1 – str1 str2) “right of string”....10
FINDC$ (char – pos|-1) (ss: --) “find character in string”....10
FIND$ (start – pos|-1) (ss: –) “find string in string”....11
REPLACE$ (-- pos) (found: ss: s1 s2 s3 -- s4 not found: s1 s2 -- s1 s2) “replace string”....11
.$ (--) (ss: str –) “display string”....11
REV$ (--) (ss: s1 – s2) “reverse string”....11
LTRIM$ (--) (ss: str1 – str2) “trim left of string”....12
RTRIM$ (--) (ss: str1 – str2) “trim right of string”....12
TRIM$ (--) (ss: str1 – str2) “trim string”....12

UCASE$ (--) (ss: str1 – str2) “convert to upper case” 12
LCASE$ (--) (ss: str1 – str2) “convert to lower case” 12
==$? (-- flag) (ss: --) “is equal to string?” 13
PICK$ (index --) (ss: -- str) “pick string” 13
$.s (--) (ss: --) 13
DEPTH$ (-- n) (ss: --) 14
The String Stack 15
Error Checking 15
String Stack Format 15
String Constant Format 15
Throw Codes 16
Dependencies 16
Author Information 16
Portable String Library Source Code 17

Introduction - The Concepts behind the Library

The String Library offers two types of strings:

- Transient strings – these exist on a string stack, which is separate from the data and return stacks. Their size is variable, and may be increased and decreased in size as necessary.
- String constants – declared with a maximum size, string constants are generally initialised to a constant string value throughout the life of the application. It is possible to change the string assigned to a string constant, but its maximum size may not be changed.

Conventions

The following coding-style conventions are employed in the library:

- Words intended to be called by a user of the library all end with a dollar sign. The dollar sign should be read as the word "string". For example, `DROP$` would be pronounced "drop string".
- Low-level words internal to the library for housekeeping, or general factors of code are surrounded with parenthesis. For example: `(lenOf$)`.

Stack Notation

Normal Forth stack notation conventions are used. Where words have an effect on the string stack, the string stack effects are shown alongside the normal data stack effects.

For example:

```
VAL$ ( -- ud ) ( ss: str -- )
```

The above example indicates that the word `VAL$` takes a string from the string stack and results in an unsigned double being pushed to the data stack.

The suggested pronunciation of the word is also given.

Acknowledgements

Whilst the code presented here is original, the concepts used in it are based on concepts developed by Brian Fox, who developed a string stack library originally for TI-Forth, and also HsForth for DOS, circa 1988. Brian was kind enough to correspond with me on the subject of string stacks, and kindly shared his code. This author extends his sincere thanks to Brian for his generosity.

String Constant Words

Since only a handful of words are associated with string constants, they will be documented first.

$CONST (max_len tib:"name" --) (runtime: -- $Caddr) "string constant"

The word `$const` declares a string constant. Declared at compile time, string constants require a maximum length and a name. For example:

```
50 $const welcome
```

The above example declares a string with a maximum size of 50 characters. It shall be referenced in code using the name welcome.

Note the runtime stack effect. It can be seen that at run-time when the name with the string is referenced it shall push its address to the data stack. The label `$Caddr` indicates that it is the address of a string constant. String constants push the address of their maximum length field which can be read with the word `$maxLen`.

MAXLEN$ ($Caddr -- max_len) "maximum length of string"

Given the address of a string constant on the data stack the word `$maxLen` returns the maximum allowed string length for that string constant.

For example:

```
50 $const welcome
welcome maxLen$ .
```

The above code fragment shall display the value 50.

:=" ($Caddr tib:"string" --) "assign string constant"

Given the address of a string constant on the data stack, the word :=" initialises the string constant with the string from the terminal input buffer.

For example:

```
50 $const welcome
welcome :=" hello and welcome!"
```

.$CONST ($Caddr --) "display string constant"

Given the address of a string constant on the data stack the word `.string` shall display the string.

For example:

```
50 $const welcome
welcome :=" hello and welcome!"
welcome .$const
```

CLEN$ ($Caddr - len) “string constant length”

Given the address of a string constant on the data stack the word `clen$` returns its actual length on the data stack. The word `$maxLen` can be used to determine the maximum length of a string constant.

For example:

```
50 $const welcome
welcome :=” hello and welcome!”
welcome clen$ .
```

The above code displays 18 – the length of the string.

>$ ($Caddr --) (ss: -- str) “to string stack”

Given the address of a string constant on the data stack the word `>$` copies the contents of the string to the string stack where it can be manipulated.

For example:

```
50 $const welcome
welcome :=” hello and welcome!”
welcome >$
```

Note that the string stack has received a copy of the string contained within `fred`. The string `fred` still exists as a string constant.

String Stack Words

The convention within this document is to refer to words that exist on the string stack as transient strings. They are referred to as transient strings because they generally only exist for a short time on the string stack. Strings are placed on the string stack and then manipulated in some way before being consumed. Memory allocation and de-allocation is managed by virtue of the strings being on the stack in the same way that the size of the data stack is managed by simply adding or removing values on the data stack.

$" (tib:"string" --) (ss: -- str) "string to string stack"

The word `$″` takes a string from the terminal input buffer and pushes it to the string stack. The end of the string is indicated by a quotation mark.

For example:

```
$″ Hello, World!″
```

In this example the string "Hello, world!" is pushed directly to the string stack, thus becoming the top item on the string stack.

Note that `$″` is a state-smart word. It can be used in both colon definitions and also directly at the command line. The correct action will be taken in either case.

In order that the *run-time* actions of `$″` may be compiled into a definition if so desired, the run-time action of this word is encapsulated within the word `($″)`. Therefore if the run-time behaviour of this word is to be compiled into another word one must compile, or postpone, the word `($″)`.

DUP$ (--) (ss: s1 -- s1 s1) "duplicate string"

The word `$DUP` duplicates the top item on the string stack.

For example:

```
$″ Hello, World!″ DUP$
```

The string stack now contains two copies of the string.

DROP$ (--) (ss: str --) "drop string"

The word `$drop` removes the topmost string item from the string stack.

For example:

```
$″ Hello, World!″
$″ How are you?″
DROP$
```

At this point the string "Hello, World!" is the topmost string the string stack. "How are you?" *was* pushed onto the string stack, but it was immediately dropped.

SWAP$ (--) (ss: s1 s2 -- s2 s1) "swap string"

The word `swap$` swaps the topmost two strings on the string stack.

For example:

```
$" Hello, World!"
$" How are you?"
```

At this point the string how are you? Is the topmost string on the string stack. If `swap$` is executed the two strings are exchanged on the string stack:

```
$" Hello, World!"
$" How are you?"
$SWAP
```

The strings are exchanged.

NIP$ (--) (ss: s1 s2 -- s2) "nip string"

The word `$nip` removes the string underneath the topmost string from the string stack.

For example:

```
$" red"
$" blue"
```

At this point, "blue" is on the top of the string stack, with "red" underneath it.

```
$NIP
```

At this point, "red" has been removed from the string stack. "blue" is the topmost string.

OVER$ (--) (ss: s1 s2 - s1 s2 s1) "over string"

The word `OVER$` pushes a copy of the string s1 to the top of the string stack, above s2.

For example:

```
$" red"
$" green"
$OVER
```

At this point, the string stack contains the following strings:

"red" (the topmost string)
"green"
"red"

ROT$ (--) (ss: s3 s2 s1 -- s2 s1 s3) \ "rotate strings"

The word ROT$ rotates the top three strings to the left. The third string (prior to the execution of ROT$) moves to the top of the string stack.

Note: For ease of implementation, this routine copies (using PICK$) the strings to the top of the string stack in their correct final order, then removes the 3 original strings underneath. Consequently, it is possible to run out of string stack space. If this happens, the condition will be correctly caught in (set$SP).

-rot$ (--) (ss: s3 s2 s1 -- s1 s3 s2) \ "rotate strings"

The word –ROT$ rotates the top three strings to the right. The top string (prior to the execution of –ROT$) moves to the third position. Note: For ease of implementation, this routine copies (using PICK$) the strings to the top of the string stack in their correct final order, then removes the 3 original strings underneath. Consequently, it is possible to run out of string stack space. If this happens, the condition will be correctly caught in (set$SP).

>$CONST ($Caddr --) (ss: str --) “to string constant”

The word `$>` takes the topmost string from the string stack and moves it into the string constant who’s address is on the data stack.

For example:

```
4 $const colour
$” red” colour $>
```

At this point, the string constant colour has the value “red”.

+$ (--) (ss: str1 str2 - str2&str1) ”concatenate strings”

The word `+$` replaces the top two strings on the string stack with their concatenated equivalent.

For example:

```
$” red”  $” blue”  +$
```

At this point, “red” and “blue” have been removed from the string stack. The topmost string on the string stack has the value “bluered”. Note that the topmost string goes to the left of the newly concatenated string.

LEN$ (-- len) (ss: --) “length of string”

The word `len$` returns the length of the topmost string on the string stack.

For example:

```
$” hello world!”  len$ .
```

Displays the value 12.

MID$ (start end --) (ss: str1 - str1 str2) "mid-string"

The word `mid$` produces a sub-string on the string stack, consisting of the characters from the topmost string starting at character *start* and ending at character *end*.

For example:

```
$" redgreenblue"  3 5 mid$
```

At this point, the topmost two strings on the string stack are as follows:

"green" (the topmost item)
"redgreenblue"

Note, as indicated in the string stack signature, the original string (str1) is retained. Note also that the first character in the string (the leftmost character) is character number 0.

LEFT$ (len --) (ss: str1 - str2) "left of string"

The word `left$` pushes the leftmost *len* characters to the string stack as a new string. The original string is retained.

For example:

```
$" redgreenblue" 3 left$
```

The above causes the string "red" to be pushed to the string stack.

RIGHT$ (len --) (ss: str1 - str1 str2) "right of string"

The word `right$` cause the rightmost *len* characters to be pushed to the string stack as a new string. The original string is retained.

For example:

```
$" redgreenblue" 4 right$
```

The above causes the string "blue" to be pushed to the string stack.

FINDC$ (char - pos|-1) (ss: --) "find character in string"

The word `findc$` returns the position of the first occurrence of the character char, beginning at the left side of the topmost string, with the search proceeding towards the right. If the character is not found, -1 is returned.

For example:

```
$" redgreenblue" char b  findc$ .
```

Displays the value 8, as the character b is found in the 8th character position (where the first character is character 0).

FIND$ (start - pos|-1) (ss: -) "find string in string"

The word `finds$` searches the second string on the string stack, starting from position *start*, for the first occurrence of the topmost string and pushes its starting position to the data stack. As a convenience, to make subsequent searches for the same substring easier, both strings are retained on the string stack.

For example:

```
$" redgreenbluegreen" $" green"  0 find$ .
```

Displays the value 3, as the substring is found at character position 3 (the leftmost character being character 0). The strings "redgreenbluegreen" and "green" remain on the stack, thus, the second instance of "green" could be found if desired.

REPLACE$ (-- pos) (found: ss: s1 s2 s3 -- s4 not found: s1 s2 -- s1 s2) "replace string"

The word replace$ searches string s2 for the first occurance of string s3. If it is found:

- it is replaced with the string s1;
- The position of s3 within s2 is pushed to the data stack;
- S1 and s3 are removed from the string stack;
- The new string is left on the string stack.

If the search string (s3) is not found:

- -1 is pushed to the data stack;
- S1 and s2 are left on the string stack, ready for another search if desired.

.$ (--) (ss: str -) "display string"

The word `.$` pops the topmost string from the string stack and displays it.

For example:

```
$" Hello, World!" .$
```

The above code displays the string "Hello, World!" on the output device.

REV$ (--) (ss: s1 - s2) "reverse string"

The word `rev$` replaces the topmost string on the string stack with its reversed equivalent.

For example:

```
$" green" rev$ .$
```

The above displays “neerg”.

LTRIM$ (--) (ss: str1 - str2) “trim left of string”

The word `$ltrim` removes leading spaces from the topmost string.

For example:

```
$"        hello!" ltrim$ .$
```

Displays “hello!” with the leading spaces removed.

RTRIM$ (--) (ss: str1 - str2) “trim right of string”

The word `$rtrim` removes leading spaces from the topmost string.

For example:

```
$" hello!      " rtrim$ .$
```

Displays “hello!” with the trailing spaces removed.

TRIM$ (--) (ss: str1 - str2) “trim string”

The word `$trim` removes both leading and trailing spaces from the topmost string.

For example:

```
$"         hello!      " $trim .$
```

The above code removes leading and trailing spaces and displays the string.

UCASE$ (--) (ss: str1 - str2) “convert to upper case”

The word `$ucase` converts all lower case characters in the topmost string to upper case.

For example:

```
$" hello world! 1234" ucase$ .$
```

The above displays “HELLO WORLD! 1234”

LCASE$ (--) (ss: str1 - str2) “convert to lower case”

The word `lcase$` converts all upper case characters in the topmost string to lower case.

For example:

```
$" HELLO WORLD! 1234" lcase$ .$
```

The above displays "hello world! 1234"

==$? (-- flag) (ss: --) "is equal to string?"

The word `$str=` performs a case-sensitive comparison of the topmost two strings on the string stack and returns true if both their length and content is identical. If the lengths or the contents differ, false is returned. The strings are retained.

For example:

```
$" hello" $" HELLO" ==$? .
```

Displays 0 (false) since the strings are different (the comparison is case sensitive).

```
$" hello" $" hello" ==$? .
```

Displays -1 (true) since the strings are identical.

```
$" hello" $" hell" ==$? .
```

Displays false, since their lengths differ.

A case in-sensitive comparison can easily be built as follows:

```
: same$? ( -- flag ) ( ss: -- )
      over$ over$ lcase$ swap$ lcase$ ==$? ;
```

PICK$ (index --) (ss: -- str) "pick string"

Given the index of a string on the string stack, copy the indexed string to the top of the string stack. `0 $pick` is equivalent to `DUP$`, `1 $pick` is equivalent to `OVER$` etc.

For example:

$" blue"
$" green"
$" red"
2 pick$

The above causes the string "blue" to be copied to the top of the string stack.

$.s (--) (ss: --)

The word `$.s` displays a non-destructive string stack dump to the output device. The length of each string is given, along with the total number of strings on the string stack. The amount of space allocated to the string stack, the amount of space in use, and the amount of free space is also reported.

DEPTH$ (-- n) (ss: --)

Returns the current depth of the string stack, with 0 meaning the string stack is empty.

The String Stack

The string stack is ALLOTED from dictionary space. The constant $sSize determines the amount of space reserved.

Error Checking

Error checking is included in all words that could cause a string stack under or overflow condition. In the event that an under or overflow is detected, the code aborts with an error message.

Other words such as DUP$ also perform checks. For example, DUP$ check that there is at least one item on the string stack. SWAP$ checks that there are at least two items on the string stack, etc.

String Stack Format

The string stack grows from higher memory addresses to lower memory addresses.

The format of the strings on the string stack is very simple, as follows:

Actual length (1 cell)	String payload (1 char=1 byte)

String Constant Format

String Constants have the same format, but are preceded by a maximum length cell in order to check that a requested string can be accommodated within the string constant:

Maximum length (1 cell)	Actual length (1 cell)	String payload (1 char=1 byte)

Throw Codes

The words in the library perform sanity checks on input parameters where necessary. In particular, the string stack, being statically ALLOTed from dictionary space, is carefully guarded, since the string stack is very likely to have code and/or data on either side of it, resulting in catastrophic software failure in the event of a string stack under or over flow. Where errors are detected, the library throws the following THROW codes, using the using ANS standard CATCH and THROW mechanism.

It should be noted that this author has not checked that the THROW codes listed here are used in other systems or libraries elsewhere.

Throw Code	Nature of Error	Thrown By
9900	String stack underflow	`DROP$` `SETS$P`
9901	String too large to assign	`:="`
9902	String stack is empty	`PICK$` `DUP$` `LEN$` `>$CONST` `MID$` `LEFT$` `RIGHT$` `FINDC$` `.$` `REV$` `LTRIM$` `RTRIM$` `UCASE$` `LCASE$`
9903	Need at least 2 strings on string stack	`SWAP$` `NIP$` `OVER$` `+$` `FIND$` `==$?`
9904	String too large for string constant	`>$CONST`
9905	Illegal LEN value	`MID$` `LEFT$` `RIGHT$`
9906	Need at least 3 strings on string stack	`ROT$` `-ROT$` `REPLACE$`
9907	String is not a legal number	`VAL$`

Dependencies

The following environmental dependencies are declared:

Word	ANS Library	ANS Reference	Referenced In
`-ROT`	None ANS. Defined as follows: `: -ROT ( a b c – c b a ) ROT ROT ;`		`:="`
`.R`	Core Ext	6.2.0210	`$.S`
`HERE`	Core	6.1.1650	`SWAP$` `+$` `REV$` `LTRIM$` `REPLACE$`
`PARSE`	Core Ext	6.2.2008	`:="` `$"`
`PICK`	Core Ext	6.2.2030	`FINDC$`
`WITHIN`	Core Ext	6.2.2440	`UCASE$` `LCASE$`

Author Information

The library was developed by Mark Wills in February 2014. The code is hereby released to the public domain. The author can be contacted by email via: markwills1970@gmail.com

Portable String Library Source Code

The source code for the string library is presented below. The library has been tested with the following systems:

- VFX Forth from MPE Ltd, United Kingdom;
- SwiftForth, from From Forth Inc., USA;
- GForth a GPL Forth.

```
\ Portable, Stack Based String Library for ANS Forth. Version 1.0
\ Mark Wills February 2014. Public Domain.

\ string format:
\   String constants (held in STRING types):
\   max_len  actual_len  <string_data>  <?>
\      |           |           |          |
\     cell        cell       chars     padding (if required)

\   Transient strings (held on the string stack):
\   actual_len  <string_data>  <?>
\      |           |             |
\     cell        chars       padding (if required)

\ Environmental dependancy declarations:
\ Word     | ANS Library | ANS Ref  | Dependencies
\ ---------+-------------+----------+------------------------------
\   -ROT   |     N/A     |   N/A    | :="
\   .R     | core ext    | 6.2.0210 | $.S
\   HERE   | core        | 6.1.1650 | SWAP$ +$ REV$ LTRIM$ REPLACE$
\   PARSE  | core ext    | 6.2.2008 | :=" $"
\   PICK   | core ext    | 6.2.2030 | FINDC$
\   WITHIN | core ext    | 6.2.2440 | UCASE$ LCASE$
\
\ Note: The word -ROT is not an ANS word. It can be defined in terms of ANS
\       words as follows: : -ROT ( a b c -- c b a ) ROT ROT ;

\ Throw codes used by this library:
\ Throw Code|Nature of Error
\ ----------+----------------------------------------
\    9900   | String stack underflow
\    9901   | String too large to assign
\    9902   | String stack is empty
\    9903   | Need at least 2 strings on string stack
\    9904   | String too large for string constant
\    9905   | Illegal LEN value
\    9906   | Need at least 3 strings on string stack
\    9907   | String is not a legal number
\    9908   | Illegal start value

base @ \ save systems' current number base
decimal

\ Set up string stack. The stack grows towards lower memory addresses.
256 \ maximum string stack size in bytes.
    \ Adjust to your own needs. Choose a value that is a multiple of your
    \ systems' cell size.

constant ($sSize)           \ store stack size
here ($sSize) allot         \ reserve space for string stack
constant ($sEnd)            \ bottom of string stack

variable ($sp)              \ pointer to top of string stack
($sEnd) ($sSize) + ($sp) ! \ initialise it
variable ($depth)           \ count of items on the string stack

variable ($temp0)           \ reserved for internal use
variable ($temp1)           \ reserved for internal use
variable ($temp2)           \ reserved for internal use
variable ($temp3)           \ reserved for internal use

\ General Note:
\ Words surrounded by parenthesis are for low-level internal use by the string
\ library, and should not need to be called by higher-level application code.

: ($depth+) ( -- )
    \ Increments the string stack item count
    1 ($depth) +! ;

: ($sp@) ( -- addr ) \ "string stack pointer fetch"
    \ Returns address of current top of string stack
    ($sp) @ ;

: ($rUp) ( n -- n|n+1)
    \ Rounds n up to the next even value
    1+ -2 and ;

: (sizeOf$) ( $addr - $size)
    \ Given an address of a transient string, compute the stack  size in bytes
    \ required to hold it, rounded up to the nearest cell size, and including
    \ the length cell.
    @ ($rUp) cell+ ;

: (set$SP) ( $size -- )
    \ Given the stack size of a transient string set the string stack pointer
    \ to the new address required to accomodate it.
    negate dup ($sp@) + ($sEnd) < if 9900 throw then
    ($sp) +! ;

: (addrOf$) ( index -- addr )
    \ Given an index into the string stack, return the start address of the
    \ string. addr points to the length cell. Topmost string is index 0,
    \ next string is index 1 and so on.
    ($sp@) swap dup if 0 do dup (sizeOf$) + loop else drop then ;

: (lenOf$) ( $addr -- len )
    \ Given the address of a transient string on the string stack (the address
    \ of the length cell), return the length of the string.
    \ Note: Immediate, compiling word for performance reasons.
    \       Modern compilers will inline this.
    state @ if postpone @ else @ then ; immediate

: depth$ ( -- $sDepth)
    \ Returns the depth of the string stack.
    ($depth) @ ;
```

```
: $const ( max_len tib:"name" -- ) ( runtime: -- $Caddr) \ "string constant"
    \ Creates a string constant. When "name" is referenced the address of the
    \ max_len field is pushed to the stack.
    \ e.g. 100 string msg
    \ The above creates a string called msg with capacity for 100 characters.
    create  dup ( max_len) , ( actual_len) 0 ,  allot align ;

: clen$ ( $Caddr -- len ) \ "string constant length"
    \ Given the address of a string constant, returns its length.
    cell+ @ ;

: maxLen$ ( $Caddr -- max_len ) \ "maximum length of string"
    \ Given the address of a string constant, returns its maximum length.
    \ Dependencies: (lenOf$)
    (lenOf$) ;

: .$const ( $Caddr -- ) \ "display string constant"
    \ Displays the string constant. e.g. fred .$const
    \ Dependencies: (lenOf$)
    cell+ dup (lenOf$) swap cell+ swap type ;

: :=" ( $Caddr tib:"string" -- ) \ "assign string constant"
    \ Assigns the string "string" to the string constant.
    \ e.g. msg :=" hello mother!"
    \ Dependencies: PARSE (core ext, 6.2.2008)
    dup @ [char] " parse swap >r
    2dup < if 9901 throw then
    nip 2dup swap cell+ !
    >r [ 2 cells ] literal + r> r> -rot cmove ;

: ($") ( addr len -- ) ( ss: -- str )
    \ Run-time action for $" (see below).
    \ Dependencies: ($rUp) ($set$SP) ($sp) ($depth+)
    dup ($rUp) cell+ (set$SP)
    dup ($sp@) !  ($sp@) cell+ swap cmove  ($depth+) ;

: $" ( tib:"string" -- ) ( ss: -- str) \ "string to string stack"
    \ Pushes a string directly to the string stack.
    \ e.g. $" hello world" .$
    \ Dependencies: ($") PARSE (core ext, 6.2.2008)
    \ Note: State smart word. Runtime behaviour is in ($")
    state @ if
        postpone s"  postpone ($")
    else
        [char] " parse  ($")
    then ; immediate

: >$ ( $Caddr -- ) ( ss: -- str) \ "to string stack"
    \ Moves a string constant to the string stack
    \ e.g. msg >$
    \ Dependencies: (lenOf$) ($")
    cell+ dup (lenOf$) swap cell+ swap ($") ;

: pick$ ( n -- ) ( ss: -- strN) \ "pick string"
    \ Given an index into the string stack, copy the indexed string to the top
    \ of the string stack.
    \ 0 $pick is equivalent to $DUP
    \ 1 $pick is equivalent to $OVER etc.
    \ Dependencies: (lenOf$) depth$ ($addrOf$) ($")
    depth$ 0= if 9902 throw then
    (addrOf$) dup (lenOf$) swap cell+ swap ($") ;

: dup$ ( -- ) ( ss: s1 -- s1 s1) \ "duplicate string"
    \ Duplicates a string on the string stack.
    \ Dependencies: depth$ pick$
    depth$ 0= if 9902 throw then
    0 pick$ ;

: drop$ ( -- ) ( ss: str -- ) \ "drop string"
    \ Drops the top string from the string stack.
    \ Dependencies: depth$ (sizeOf$) (set$SP)
    depth$ 0= if 9900 throw then
    ($sp@) (sizeOf$) negate (set$SP)   -1 ($depth) +! ;

: swap$ ( -- ) ( ss: s1 s2 -- s2 s1) \ "swap string"
    \ Swaps the top two string items on the string stack.
    \ Dependencies: depth$ (sizeOf$) (addrOf$) HERE (core 6.1.1650)
    depth$ 2 < if 9903 throw then
    ($sp@) dup (sizeOf$) here swap cmove
    1 (addrOf$) dup (sizeOf$) ($sp@) swap cmove
    here dup (sizeOf$)  ($sp@) dup (sizeOf$) + swap cmove ;

: nip$ ( -- ) ( ss: s1 s2 -- s2) \ "nip string"
    \ Remove the string under the top string.
    \ Dependencies: swap$ drop$ depth$
    depth$ 2 < if 9903 throw then
    swap$ drop$ ;

: over$ ( -- ) ( ss: s1 s2 -- s1 s2 s1) \ "over string"
    \ Move a copy of s1 to top of string stack.
    \ Dependencies: pick$ depth$
    depth$ 2 < if 9903 throw then
    1 pick$ ;

: (rot$) ( -- ) ( ss: s6 s5 s4 s3 s2 s1 -- s3 s2 s1)
    \ move the top three strings downwards by three strings
    \ internal factor of both ROT$ and -ROT$
    \ Dependencies: ($sp@) (sizeOf$) (addrOf$) ($sp) ($depth)
    ($sp@) (sizeOf$)  1 (addrOf$) (sizeOf$)  2 (addrOf$) (sizeOf$) + +  cmove
    3 (addrOf$) ($sp) !  -3 ($depth) +! ;

: rot$ ( -- ) ( ss: s3 s2 s1 -- s2 s1 s3) \ "rotate strings"
    \ Rotates the top three string to the left.
    \ The third string moves to the top of the string stack.
    \ Note: For ease of implementation, this routine copies (using PICK$)
    \ the strings to the top of the string stack in their correct final
    \ order, then removes the 3 original strings underneath.
    \ Consequently, it is possible to run out of string stack space.
    \ If this happens, the condition will be correctly caught in (set$SP).
    \ Dependencies: pick$ (rot$) depth$
    depth$ 3 < if 9906 throw then
    1 pick$  1 pick$  4 pick$ (rot$) ;

: -rot$ ( -- ) ( ss: s3 s2 s1 -- s1 s3 s2) \ "rotate strings"
    \ Rotates the top three string to the right.
    \ The top string moves to the third position.
    \ Note: For ease of implementation, this routine copies (using PICK$)
    \ the strings to the top of the string stack in their correct final
    \ order, then removes the 3 original strings underneath.
    \ Consequently, it is possible to run out of string stack space.
    \ If this happens, the condition will be correctly caught in (set$SP).
    \ Dependencies: pick$ (rot$) depth$
    depth$ 3 < if 9906 throw then
    0 pick$  3 pick$  3 pick$ (rot$) ;

: len$ ( -- len ) ( ss: -- ) \ "length of string"
    \ Returns the length of the topmost string.
    \ Dependencies: none
    depth$ 1 < if 9902 throw then
    ($sp@) @ ;

: >$const ( $Caddr -- ) ( ss: str -- ) \ "to string constant"
    \ Move top of string stack to the string constant.
    \ e.g. $" blue" fred >$const  fred .$const
    \ displays "blue"
    \ Dependencies: depth$ (sizeOf$) drop$
    >r  depth$ 1 < if 9902 throw then
    len$ r@ @ > if 9904 throw then
    ($sp@) dup (sizeOf$) r> cell+ swap cmove drop$ ;

: +$ ( -- ) ( ss: s1 s2 -- s2+s1) \ concatenate strings
    \ Replaces the top most two strings on the string stack with their
    \ concatenated equivalent.
    \ eg: $" red" $" blue" +$ .$
    \ displays "redblue"
    \ Dependencies: depth$ (addrOf$) (lenOf$) len$ drop$ HERE (core 6.1.1650)
    depth$ 2 < if 9903 throw then
    1 (addrof$) cell+  here   1 (addrof$) (lenof$)  cmove
    ($sp@) cell+   1 (addrof$) (lenof$) here +  len$ cmove
    here len$ 1 (addrof$) (lenof$) +  drop$ drop$  ($") ;

: mid$ ( start len -- ) ( ss: str1 -- str1 str2) \ "mid-string"
    \ The characters from start to start+len are pushed to the string stack
    \ as a new string. The original string is retained.
    \ Dependencies: len$ ($")
    depth$ 1 < if 9902 throw then
    dup len$ >  over 1 < or  if 9905 throw then
    over dup len$ >  swap 0< or if 9908 throw then
    swap ($sp@) cell+ +  swap  ($") ;

: left$ ( len -- ) ( ss: str1 -- str1 str2) \ "left of string"
    \ The leftmost len characters are pushed to  the string stack as a new
    \ string. The original string is retained.
    \ Dependencies: mid$
    depth$ 1 < if 9902 throw then
    dup len$ > over 1 < or if 9905 throw then
    0 ($sp@) cell+ +  swap  ($") ;

: right$ ( len -- ) ( ss: str1 -- str1 str2) \ "right of string"
    \ The rightmost len characters, pushed to the string stack as a new string.
    \ the original string is retained.
    \ Dependencies: (lenOf$) mid$
    depth$ 1 < if 9902 throw then
    dup len$ > over 1 < or if 9905 throw then
    ($sp@) (lenOf$) over - ($sp@) cell+ +  swap  ($") ;
```

```
: findc$ ( char -- pos|-1 ) ( ss: -- ) \ "find character in string"
    \ Returns the first occurance of the character char in  the top string.
    \ The string is retained. Returns -1 if the char is not found.
    \ Dependencies: PICK (ANS core ext) depth$
    depth$ 1 < if 9902 throw then
    ($sp@) cell+  ($sp@) (lenOf$) 0 do
        dup c@ 2 pick = if i -1 leave then 1+ loop
     -1 = if nip nip else drop -1 then ;

: find$ ( offset -- pos|-1 ) ( ss: s1 s2 -- s1) \ "find string"
    \ Searches string s1, beginning at offset, for the substring s2.
    \ If the string is found, returns the position of the string relative
    \ to the offset, otherwise returns -1.
    \ Dependencies: depth$ len$ (addrOf$) (lenOf$) drop$
    depth$ 2 < if 9903 throw then
    len$ ($temp1) !    1 (addrOf$) (lenOf$) ($temp0) !
    dup ($temp0) @ > if drop -1 exit then
    1 (addrOf$) cell+ + ($temp2) !    ($sp@) cell+ ($temp3) !
    ($temp1) @ ($temp0) @ > if drop -1 exit then
    0  ($temp0) @ 0 do
        ($temp3) @ over + c@
        ($temp2) @ i + c@ = if
            1+ dup ($temp1) @ = if
                drop i ($temp1) @ - 1+   -2 leave then
        else drop 0 then
    loop
    dup -2 = if drop else drop -1 then drop$ ;

: .$ ( -- ) ( ss: str -- ) \ "display string"
    \ Pop and display the topmost string from string stack.
    \ Dependencies: depth$ (lenOf$) drop$
    depth$ 0= if 9902 throw then
    ($sp@) cell+ ($sp@) (lenOf$) type  drop$ ;

: rev$ ( -- ) ( ss: s1 -- s2 ) \ "reverse string"
    \ Reverse topmost string on string stack.
    \ Dependencies: depth$ (lenOf$) HERE (core 6.1.1650)
    depth$ 0= if 9902 throw then
    ($sp@) dup cell+ >r  (lenOf$)  r> swap here swap cmove
    ($sp@) (lenOf$) here 1- +
    ($sp@) cell+  dup ($sp@) (lenOf$) +   swap do
        dup c@ i c!  1- loop  drop ;

: ltrim$ ( -- ) ( ss: s1 -- s2 ) \ "left trim string"
    \ Removes leading spaces from s1, resulting in s2.
    \ Dependencies: depth$ (lenOf$) (sizeOf$) drop$ HERE (core 6.1.1650)
    depth$ 0= if 9902 throw then
    ($sp@) dup (lenOf$) >r  here over (sizeOf$)  cmove
    0  r> here cell+ dup >r +  r> do
        i c@ bl = if 1+ else leave then loop
    dup 0> if
        >r  ($sp@) (lenOf$)  drop$
        here cell+ r@ +  swap r> -  ($")
    else drop then ;

: rtrim$ ( -- ) ( ss: s1 -- s2 ) \ "right trim string"
    \ Removes trailing spaces from s1, resulting in s2.
    \ Dependencies: depth$ rev$ ltrim$
    depth$ 0= if 9902 throw then
    rev$ ltrim$ rev$ ;

: $trim ( -- ) ( ss: s1 -- s2 ) \ "trim string"
    \ Remove both leading and trailing spaces from s1, resulting in s2.
    \ Dependencies: rtrim$ ltrim$
    rtrim$ ltrim$ ;

: replace$ ( -- pos ) ( found: ss: s1 s2 s3 -- s4  not found: s1 s2 -- s1 s2)
    \ In string s2 find s3 and replace with s1, resulting in s4.
    \ If a replacement is made, the starting position of the replacement is
    \ returned, otherwise -1 is returned.
    \ Dependencies: depth$ find$ (addrOf$) (lenOf$) drop$ ($")
    \               nip$ HERE (core 6.1.1650)
    depth$ 3 < if 9906 throw then
    len$ >r
    0 find$ dup ($temp0) ! -1 > if
        ($sp@) cell+  here  ($temp0) @ cmove
        1 (addrOf$) cell+   here ($temp0) @ +
        1 (addrOf$) (lenof$) cmove
        ($sp@) cell+ ($temp0) @ + r@ +
        here ($temp0) @ + 1 (addrOf$) (lenof$) +
        len$ r> - ($temp0) @ -  dup >r  cmove
        r> ($temp0) @ + 1 (addrOf$) (lenof$) +
        drop$ drop$ here swap ($")
    else r> drop ($temp0) @ then ;

: ucase$ ( -- ) ( ss: str -- STR) \ "convert to upper case"
    \ On the topmost string, converts all lower case characters to upper case.
    \ Dependencies: WITHIN (core ext) (lenOf$) depth$
    depth$ 1 < if 9902 throw then
    ($sp@) dup (lenOf$) + cell+  ($sp@) cell+  do
       i c@ dup [ char a ] literal  [ char { ] literal within if
            32 -  i c! else drop then loop ;

: lcase$ ( -- ) ( ss: STR -- str) \ "convert to lower case"
    \ On the topmost string, converts all upper case characters to lower case.
    \ Dependencies: WITHIN (core ext) (lenOf$) depth$
    depth$ 1 < if 9902 throw then
    ($sp@) dup (lenOf$) + cell+  ($sp@) cell+  do
       i c@ dup [ char A ] literal  [ char [ ] literal within if
            32 +  i c! else drop then loop ;

: ==$? ( -- flag ) ( ss: -- ) \ "is equal to string"
    \ Performs a case-sensitive comparison of the topmost two strings on the
    \ string stack, returning true if their length and contents are identical,
    \ otherwise returning false.
    \ Dependencies: depth$ (addrOf$) (lenOf$)
    depth$ 2 < if 9903 throw then
    len$ 1 (addrOf$) (lenOf$) = if
        1 (addrOf$) cell+
        ($sp@) cell+ ($sp@) +  ($sp@) cell+ do
            dup c@  i c@  <> if drop false leave then 1+ loop
        dup if drop true then
    else false then ;

: val$ ( -- ud ) ( ss: str -- )
    \ Interprets the topmost string as an integer number, returning its value
    \ on the data stack as an unsigned double integer (see 6.1.0570)
    \ Dependencies: (lenOf$) drop$
    0 0 ( ud1) ($sp@) dup (lenOf$) swap cell+ swap ( c-addr1 u1)
    >number if 9907 throw then
    drop  drop$ ;

: ud>$ ( ud -- ) ( ss: -- str )
    \ Pushes the unsigned double number on the data stack to the string stack.
    \ Dependencies: ($")
    <# #s #> ($") ;

: $.s ( -- ) ( ss: -- )
    \ Non-destructively displays the string stack.
    \ Dependencies: depth$ len$ .$ .R (core ext, 6.2.0210)
    cr  depth$ 0> if
        ($sp@)  depth$
        ." Index|Length|String" cr
        ." -----+------+------" cr
        0 begin
            depth$ 0> while
                dup 5 .r ." |" len$ 6 .r  ." |" .$  1+ cr
        repeat  drop
        ($depth) !  ($sp) !  cr
    else
        ." String stack is empty." cr
    then
    ." Allocated stack space:" ($sEnd) ($sSize) + ($sp@) - 4 .r ."  bytes" cr
    ."     Total stack space:" ($sSize) 4 .r ."  bytes" cr
    ." Stack space remaining:" ($sp@) ($sEnd) - 4 .r ."  bytes" cr ;

base ! \ restore systems' current number base
```