A descriptor based approach to Forth strings

Ulrich Hoffmann (FH Wedel University of Applied Sciences), Andrew Read

August 2018

uh@fh-wedel.de, andrew81244@outlook.com

Abstract

We have developed a novel, descriptor based based approach to Forth strings based on the established technique of array slices, most recently exemplified by the Go programming language. We have done this with the goal of creating a strings package that is suitable be incorporated into the Forth kernel. We illustrate the convenience and potential utility of our approach with a lightweight regular expression matcher. We argue that the descriptor based approach provides much of the functionality that has been traditionally obtained with string stacks, but more simply and closely integrated with Forth, and therefore more easilty adopted as a foundational layer in the kernel. We see a potential role for our approach on any Forth system needing more than the bare minimalism of `c-addr u` strings.

1 Introduction

Forth is more than 40 years old, yet strings remain a space for innovation. We are seeking to develop a strings package that can be implemented as a foundational layer in the Forth kernel upon which the string processing requirements of the Forth kernel can be based. Our descriptor-based approach has been informed by the technique of array slices, as most recently exemplified in the Go programming language.

Our paper begins with brief examples of array slices drawn from other computer languages. Moving on to Forth, we review Anton Ertl's main observations from his EuroFORTH 2013 paper "Standardize Strings Now!" [5]. We consider some other contributions to Forth strings that have appeared over the years, before stating our goals and evaluation criteria, describing our approach and illustrating it with a lightweight regular expression matcher that we have developed from original code by Rob Pike and Brian Kernighan. We discuss the advantages and limitations of our approach and make a brief comparative analysis with strings stacks. Finally we consider the suitability of our descriptor based approach for different Forth environments.

2 Array slices

The Go programming language has both arrays and slices. Arrays are conventional data-structures holding a collection of elements of the same type. Go arrays are value types: assigning a Go array to a new variable creates a copy of the array. Slices are reference-type wrappers for arrays. Slices refer segments of an array that may be manipulated without copying or modifying the underlying data [1]. Multiple slices may reference the same array, potentially to overlapping ranges within it. Go arrays themselves have a fixed size but slices vary in size. Slice functions might create new slices, which reference a newly allocated array with larger capacity and copy over the elements. The old array might later be garbage collected if not referenced by other slices.

Other examples include: Algol 68, which permitted slices as references to subsections of arrays specified between lower and upper bounds, Fortran 77, which offered array slices with sophicticated capabilities for slicing multi-dimensional arrays in any direction, and Sinclair BASIC (ZX80/81 and ZX Spectrum), which offered simple but convenient slicing of character strings [2, 3, 4].

3 Standardize Strings Now!

Anton Ertl began "Standardize Strings Now!" by arguing that the desirable properties for Forth strings are (i) ease of use and (ii) integration with the rest of Forth.

The first part of Anton's article discussed these requirements as they relate to memory management of the character buffer. One barrier to efficient string handling is the allocation and deallocation of character buffers as strings are consumed by words on the stack. Anton illustrated the pitfalls of either ignoring the problem altogether (leading to memory leaks or to the recruitment of the ever-unpopular garbage collector) or attempting to manage allocation and deallocation concurrently with the manipulation of strings (leading to difficult code and reliance on `allocate`, a word that many small systems prefer not to support). Anton explained how region-based memory allocation can assist by collecting related data into a single region that can easily be freed as a block.

The second part of Anton's article discussed the convenient representation of strings. Anton highlights some of the current issues: the `c-addr u` format is flexible because it represents strings of arbitrary length and content and allows sub-strings to be taken without copying the buffer. A disadvantage of this approach is that it takes two cells to represent each string and this becomes cumbersome with several strings. The counted string format needs only one cell on the stack (a pointer to the count) but sub-strings cannot be taken without copying the buffer, and traditional implications of the counted cell format can only represent strings with up to 255 bytes.

A practical illustration of the stack depth difficulty with the `c-addr u` representation is illustrated by Anton's example of a regular expression matching word

```
: search-regexp ( c-a1 u1 c-a2 u2 -- c-a1 u3 c-a4 u4 c-a5 u5 true | false )
\ Search for regexp c-a2 u2 in string c-a1 u1
\ if the regexp is found, c-a1 u3 is the substring before the first match,
\ c-a4 u4 is the first match, and c-a5 u5 is the rest of the string, and
\ TOS is true, otherwise return false
```

A successful match returns 7 parameters on the stack (3 strings and a flag). Anton suggested the possibility of implicit parameters and context-wrappers among other techniques for reducing stack depth, and concluded in general by favouring the `c-addr u` format over most alternatives.

4 Other contributions

Strings stacks with special features for handling strings have been around since the early-days of Forth. For example Klaus Schliesiek's 1986 string stack implementation, later modernized and updated for Forth-94 and Forth-2012 compliant systems by Ulli Hoffmann [8]. One attraction of string stack is their capability for handling multiple strings without cluttering the parameter stack. For example:

```
: "delimiter-join (" s1 s2 ... sn delim -- s ) ( n -- )
\ Concatenate the n strings sn, ... s2, s1 with
\ sn at the beginning of the resulting string s
\ intersperced with the delimiter string delim.
\ " ef" " cd" " ab" 3 " /" "delimiter-join results in ab/cd/ef.

: "delimiter-split (" s0 delim -- s1 ... sn ) ( -- n )
\ Split the text s0 on occurances of the delimiter string delim.
\ s1 to sn are the resulting parts. sn is the closest to the beginning of s0.
\ The delimiter is removed. n is the number of parts.
```

Brad Rodriguez developed PatternForth to provide SNOBOL4-like string processing and pattern matching functions. PatternForth, in a different context, incorporates the concept of string descriptors [13]. More recently Carsten Strotmann has suggested that REXX Parse provides a worthwhile model that may be useful for inspiring future work with Forth strings [6]. Ulli Hoffmann has also demonstrated a very light wordset for string handling that represents strings entirely on the parameter stack as a count followed by characters one-cell-at-a-time. This approach that has the advantage of minimal dependency on the rest of the Forth dictionary [7].

5 Goal and evaluation criteria

Our motivation in developing this strings package is connected with the our concept of a "New Synthesis" for Forth [12]. We seek to develop an approach to strings that can be built into the Forth kernel and used to support the

string handling requirements of the kernel, such as I/O operations, input stream processing and maintenance of the Forth dictionary. The New Synthesis is a concept under development, nevertheless some criteria suggest themselves based on these ambitions.

The strings package must be programmed before the kernel is completed, so it must be developed using an elementary Forth vocabulary. The string handling facilities of the kernel will be developed out of our package, so logically we cannot rely on those facilities in developing the package itself. We do not want the strings package to commit us to fixed decisions elsewhere in the kernel, such as choices about memory management. We would prefer the compilation size of our package to be small (although not necessarily minimal, recognizing that all design decisions in a Forth system are eventually tradeoff decisions). We must ensure adequate performance, and experience suggests that we must therefore avoid any approach that relies on copying string data. Ideally there should be some "killer application" for our package within the kernel itself, and for us this is the opportunity to parse the input stream with regular expressions (see section 6). Finally, the strings package should be generally usable and useful for applications running on our kernel that we cannot yet anticipate.

6 A descriptor based approach to strings

6.1 Outline

Our strings are single-cell references on the parameter stack. Each reference (our 'string') points to a small block of string meta-data (our 'descriptor') located in memory. The descriptor, among other information, holds a flexible onward reference to an ASCII character buffer where the string data resides.

The descriptor based approach allows strings to be duplicated or sub-strings to be created without any copying of character data. We have also established two classes of strings: 'permanent' strings and 'temporary' strings, which function similarly, but which have different consequences for memory management.

Before describing our approach in more detail we provide an illustration of usage. String operators in our wordset are prefixed with `$` (e.g. `$initialize, $make, $len, $s`) while strings themselves are suffixed with `$` (`a$, pad$`, etc.).

```
\ reserve space for 10 string descriptors
10 $initialize

\ Establish an ASCII character buffer with 9 character of text
\ and 20 characters of extra capacity
S" Veni vidi012345678901234567890"

\ Create a permanent string and leave its descriptor on the stack
\ : $make ( c-addr len size flag -- s$)
\ permanent = -1
9 swap permanent $make

\ Create another string, this time a temporary string with no extra capacity
\ temporary = 0
S"  vici" dup temporary $make

\ Concatenate the two strings (see the wordlist that follows)
\ : $+ ( s$ r$ -- s$)
$+

\ Enquire as to the length of the new string
\ : $len ( s$ -- s$ n)
$len CR .
14

\ Provide a legacy reference to the string, and type it out
\ : $s ( s$ -- c-addr u)
$s CR type
Veni vidi vici
```

6.2 Overview of the descriptor approach

Figure 1 illustrates the descriptor approach. Our descriptors are structures with four cell-width fields: the address field points to the start address of the character buffer, the size field holds the total size of the character buffer

in bytes, the start field holds the position within the character buffer of the first character of the string (counted from zero for the first byte of the buffer), the length field holds the count of characters in the string. A single flag bit[1] specifies whether the string is permanent or temporary (see the following section on Memory management). Empty strings are represented by descriptors with a length field of zero.

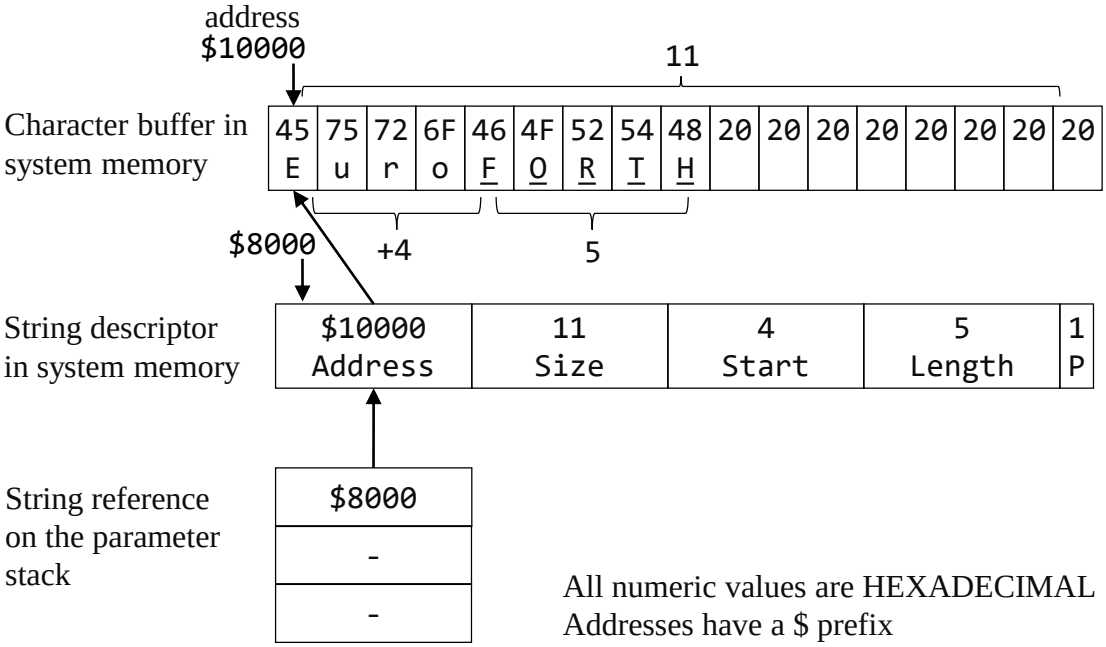

Figure 1: The descriptor approach illustrating a reference to the characters "FORTH" (underlined) held in a character buffer. In this example the permanent flag of the descriptor has arbitrarily been set to TRUE. Memory addresses and data are purely illustrative.

Strings are duplicated by creating a replica descriptor without copying any character data, as illustrated in figure 2. Sub-strings are taken by modifying the start and length fields. A string can be appended to by copying data into the character buffer and incrementing the length field, provided that the size of the character buffer has sufficient capacity. Traditional `c-addr u` references can be obtained by computing the address of the first character in the string (address + start) and also providing the length. More complex manipulations, such as inserting and deleting characters within a string, can be carried out by modifying the character buffer, again subject to the constraint of the size of the character buffer.

6.3 Memory management

Memory management needs to address two separate issues: management of the character buffers and management of the descriptors themselves.

6.3.1 Character buffers

In our implementation we explicitly do not memory manage the character buffers. To create a string descriptor it is necessary that the proposed character buffer already exist in memory. When a string descriptor is recycled (see below), the character buffer remains intact. In consequence, character buffers need to be memory managed separately. We discuss the merits and limitations of this approach later.

[1] In our implementation the permanent/temporary bit is actually the MSB of the size field, so that strings are limited in size to 2^31 bytes on a 32 bit system. We require that the MSB of the length and start fields must be zero. However different structures would be equally feasible.

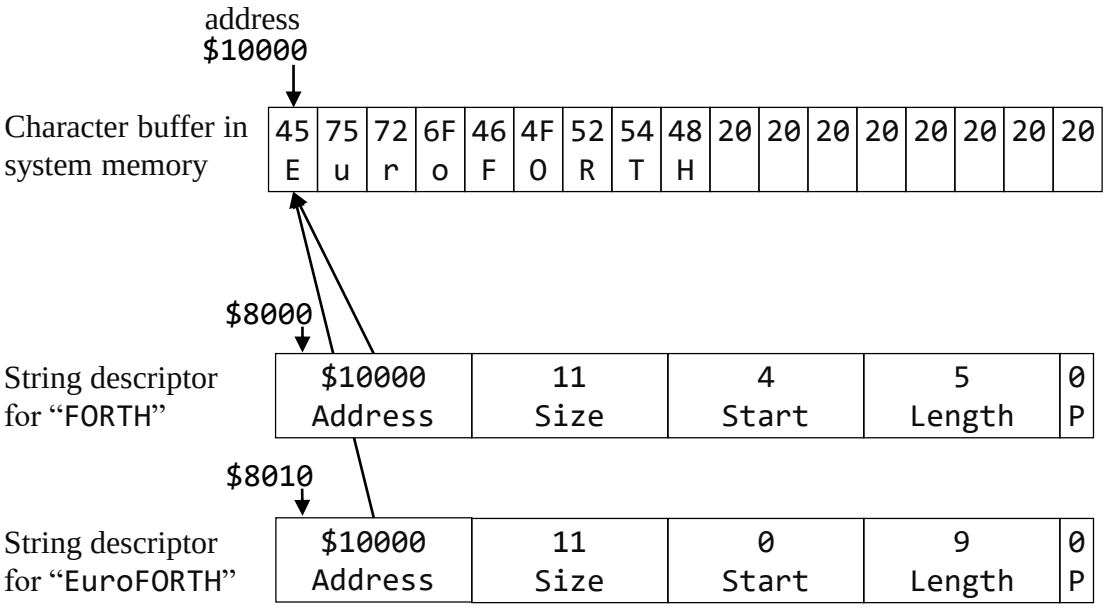

Figure 2: Two descriptors that point to different substrings in the same character buffer. The second descriptor may have been created as a duplicate of the first and later modified. Note that changes to the string buffer would affect both strings.

6.3.2 String descriptors

We reserve memory for a fixed number of string descriptors at initialization[2] (a pool approach). We need to identify those string descriptors within the pool that are unallocated and available for creating new strings. In our first iteration, we opted to use a scheme with an additional stack stack set up in memory to hold the addresses of unallocated descriptors. The inefficiency of this approach was pointed out by the anonymous academic reviewers who suggested a revised approach that we have now adopted. This is a linked list in which the address field each unallocated descriptor points to the address of the next unallocated descriptor. A global variable within our strings package holds the address of the first free descriptor and the list is terminated with a zero pointer. The descriptor pool is illustrated in figure 3.

6.3.3 Recycling descriptors

As noted above, our strings are represented by a single-cell reference on the stack. Applications may discard these references as strings are no longer required. Without an appropriate mechanism to recycle the descriptors of discarded strings the pool of available descriptors would eventually become depleted.

We differentiate between permanent and temporary strings, an assignment made when a string is created. Permanent strings are never recycled, temporary strings are subject to the recycling mechanisms described below. Having both permanent and temporary strings allows flexibility for strings that have different intended uses. Looking ahead, we might create a permanent string to refer to a repeatedly-used regular expression, whilst we might create a temporary string to refer to characters in some temporary buffer.

The recycling word is `$drop (s$ --)`. `$drop` take the string `s$` from the top of the stack and recycles it. When a string descriptor is recycled it is replaced in the pool of available descriptors. We also take the precaution of "spoiling" a recycled string (i.e. setting its size, length and start position to zero), with the intention of making it harder to inadvertently keep using a descriptor after recycling it. As noted already, when a string is recycled by `$drop` the character buffer to which it referred is left untouched. Also `$drop` detects permanent strings and does not recycle them.

[2] In our implementation we use ALLOCATE to reserve the necessary storage for string descriptors, but as this is a once-only procedure at initialization time it would be equally possible to reserve the necessary storage with ALLOT, or in some platform-dependent manner.

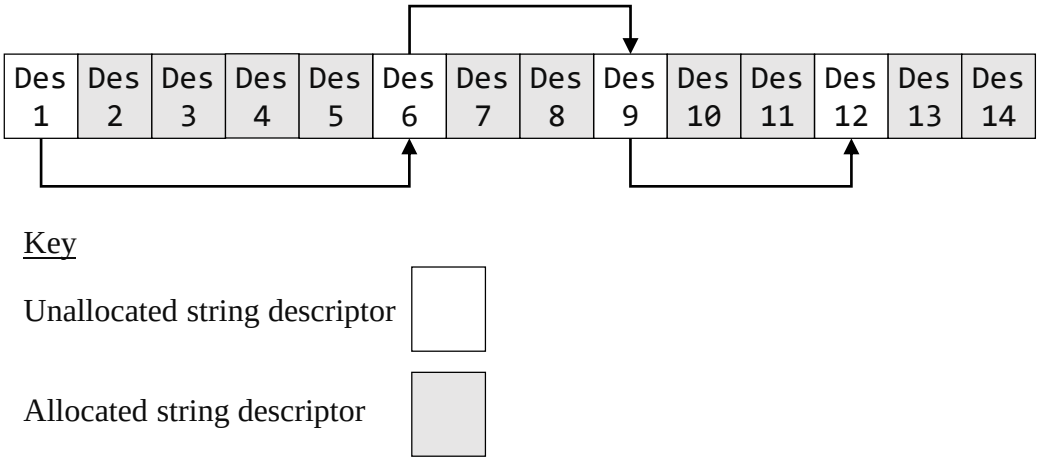

Figure 3: Illustration of the descriptor pool and stack of available descriptors. In this example 10 descriptors have been allocated out of 14 in the pool. References to the 4 unallocated descriptors are held in a linked list.

At this point it may be worth comparing the actions of `$drop` with `drop`. Both `$drop` and `drop` will, if there is a string descriptor on the top of the stack, remove that string descriptor from the top of the stack. However `$drop` will also recycle the descriptor (assuming that it was a temporary string). By contrast `drop` will not recycle the descriptor, which will remain intact.

Words in our string wordset that consume strings as parameters will recycle their descriptors. For example

```
: $+ (s$ r$ -- s$)
\ Append the contents of string r$ to s$ and return s$.
\ The length of the string is always truncated to fit within the size of s
\ r$ is internally passed to $drop for recycling
```

In cases where we do not want to consume a string parameter, the convention is that we do not consume it. For example

```
$len ( s$ -- s$ n)
\ Return the length of a string in count of characters
```

To summarize the convention that we have adopted: the recycling of a string descriptor can be forced with `$drop`, otherwise string descriptors are recycled automatically when they are consumed by a word in our string wordset. However those strings designated at the time of creation as permanent are not recycled in these cases.

6.3.4 Duplicating strings

With our approach it is possible to duplicate strings without copying the underlying character data. The duplication word is `$dup ( s$ -- s$ r$)`. `$dup` takes the string descriptor `s$` and creates a new string descriptor `r$`. `r$` has the same size, length, and start position as `s$`, and it also refers to the same character data. However r$ and s$ refer to separate, distinct descriptors.

Compare the action of `$dup` with `dup`: both `$dup` and `dup` will, if there is a string descriptor on the top of the stack, place a second string descriptor on the top of the stack. However `$dup` actually creates a new descriptor. In this case if `s$` is subsequently modified by taking a sub-string with `$sub`, or by adding another string with `$+`, or recycled with `$drop`, `r$` will not be affected. Of course `s$` and `r$` still refer to the same underlying character data and any operations that affect that character data directly (such as inserting or deleting characters) will affect both `s$` and `r$`.

drop: remove the descriptor from the stack but do not recycle it	**$drop**: remove the descriptor from the stack and recycle it (if it is a temporary string)
dup: duplicate the reference to the same descriptor	**$dup**: create a new descriptor and provide a reference to it

Table 1: Comparison of drop and $drop, and dup and $dup

Table 1 compares the actions of `drop` and `$drop`, and `dup` and `$dup`.

6.4 Wordlist

To complete the description of our descriptor based approach to strings we now list the main words currently in our wordlist.

```
: initialize ( N --)
\ Initialize the string descriptor system with space for N strings

: $make ( c-addr len size flag -- s$)
\ Make a new string descriptor referencing character data at c-addr
\ The character buffer contacts len bytes of valid data, starting at c-addr
\ and has total capacity of size bytes.
\ If size > len then the string has spare capacity to be extended
\ flag = TRUE for a permanent string; FALSE for a temporary string

: $drop ( s$ --)
\ Recycle a the descriptor s$, unless s$ is a permanent string
\ The character data itself is not deallocated

: $s ( s$ -- c-addr u)
\ provide a legacy reference to a string

: $len ( s$ -- s$ n)
\ return the length of a string in characters

: $size ( s$ -- s$ n)
\ return the size of the character buffer holding the string

: $dup ( s$ -- s$ r$)
\ Copy the string descriptor s$ to a new string descriptor r$
\ Both $s and $r reference the same character data in memory
\ but can take different cuts

: $sub ( s$ a n -- s$)
\ Modify s$ to reference the substring starting at position a
\ and running for n characters
\ a is calculated as an offset from the current start position,
\ not the start of the buffer
\ a is permitted to be negative; n should be positive

: $app ( s$ c-addr u -- s$)
\ Append the text characters from c-addr u to s$ and return the augmented string
\ The length of the string is always truncated to fit within size

: $+ ( s$ r$ -- s$)
\ Append the contents of string r$ to s$ and return s$.
\ The length of the string is always truncated to fit within size

: $= ( s$ r$ -- s$ r$ flag)
\ Compare the character strings s$ and r$ and return true if they are equal
\ Note, this compares the characters in the buffer, not the descriptors

: $rem ( s$ a n -- s$)
\ Remove n characters from s$ starting at position a
\ Following characters within the character buffer are moved as necessary

: $ins ( c-addr u s$ a -- s$)
\ Copy the text characters from c-adder u into s$ at position a
```

```
\ Following characters within the character buffer are moved as necessary
\ The length of the string is always truncated to fit within size
```

7 Illustrative application - a regular expression matcher

In section 2 we mentioned Anton Ertl's illustration of the cumbersome stack signature for a regular expression matcher and suggested solutions. The descriptor based strings approach offer an alternative and explains some of our design decisions. Thanks to single cell string references, the regular expression matcher accepts 2 parameters and returns no more than 4. Placing the length, start and size within the string descriptor allows us to partition the original string according to the results of the regular expression search without any string copying.

The regular expression matcher is less that 250 lines long, including full comments. This code originated as a direct port of a C program written in 1998 by Rob Pike and Brian Kernighan [11]. For the sake of brevity and simplicity we do not attempt to support more complex regular expressions, but the code is straightforward enough that extensions could be added by a user if needed. We have however extended the basic set of matches to include specific regular expressions focused on parsing the Forth input stream, see table 2.

The wordlist is briefly illustrated below. `$regex` calls `match`, which takes `c-addr u` strings as input parameters and returns the character location of the match as the output. `match` can be used independently of the descriptor based string library, but in that case the application must perform additional manipulation to extract the matched and unmatched portions of the string.

```
: $regex ( s$ r$ -- a$ b$ s$ TRUE | FALSE)
\ Search for regex r$ in string s$ if the regexp is found, a$ is the
\ substring before the first match, b$ is the first match
\ s$ (modified) is the rest of the string and the TOS is true;
\ otherwise return false and preserve s$ unmodified
\ r$ is $drop'ed (recycled unless defined to be a permanent string)
\ a$, b$ and s$ all reference portions of the same character data in memory

: match ( addrT uT addrR uR -- first len TRUE | FALSE )
\ search for regexp (addrR uR) anywhere in text (addrT uT)
\ return the position of the start of the match, the length of the match,
\ and TRUE or FALSE if there is no match
```

8 Discussion

8.1 Some design considerations

8.1.1 Different descriptors pointing to substrings in the same mutable string

Figure 2 illustrates two separate descriptors pointing to different, but overlapping, substrings in the same mutable string buffer. This seems like a recipe for disaster. However there are advantages if used with care, and we have one "use case" that we believe is compelling. Let the input stream be represented by a string descriptor, and proceed to parse it with our regular expression matching word, `$regex`. A successful match returns three descriptors, all of which point to non-overlapping sections of the original input stream buffer. This is very efficiently achieved without any string copying. Furthermore, the string descriptor that represents the input stream is automatically adjusted to the unmatched portion, also without any string copying. We anticipate there could be other instances within the kernel where the ability to split and alter strings without copying may be an advantage.

8.1.2 Automatic recycling of consumed string descriptors

Our approach is to automatically recycle the relevant descriptors when strings are consumed by string manipulation words. The alternative would have been to not do this and leave the user to recycle all strings at the end of their lifetime with `$drop`. The potential advantage is reduced need for string duplication with `$dup` in cases where a string will still be needed after a string manipulation word consumes it as a parameter.

We felt the over-riding consideration was to make management of the descriptors as trouble-free for the user as possible and that this was best achieved by the approach we have taken. In addition, it is a simple matter to

Regex	Match
^	Beginning of string
!	Beginning of string disregarding succeeding whitespaces
$	End of string
.	Any character, including newline
\	Quote or special character
a*	Zero or more a's
a+	One or more a's
a?	Zero or one a's (i.e. optional a)
~a	Not a (i.e. any character other than a)
\t	Tab
\n	Linefeed
\r	Carriage return
\s	Any whitespace
\S	Any non-whitespace
\d	Any decimal digit
\h	Any hexadecimal digit (case insensitive)

Table 2: Regex's supported by our regular expression matcher

arrange for a string manipulation word not to consume its argument when it is anticipated that the string will be needed again, and this is illustrated by the stack signatures of `$len` and `$size`.

We have provided for permanent strings that are totally immune to recycling. Our motivation for permanent strings is the opportunity to assign certain components within the kernel to descriptor based strings, where it would be convenient for system operations not to have them recycled. For example we anticipate building a kernel in which the input stream and regular expressions for certain input types (such as decimal and hexadecimal numbers) are represented by permanent strings.

8.2 Relationship to the string stack approach

Our approach can be compared to a non-copying string stack as illustrated in figure 4. The similarity is that the string is ultimately represented by an abstract datatype that holds more information than simply the length and address of the string. In string-stack system these datatypes reside on the stack directly. In our system the abstract datatypes have been placed in a pool of descriptors, while the references to those descriptors are placed on the parameter stack.

It can be argued that the string descriptor approach is simpler and better integrated with Forth. Firstly, there is no need for a separate string stack with its own suite of stack handling words. Secondly, both string and non-string arguments can combined in the signatures of string-handling words. The "secret weapon" of the descriptor-based approach that gives string-stack like functionality on the parameter stack is the subtle differences in operation between `$dup` and `dup`, and `$drop` and `drop`.

Another relative convenience of the descriptor based approach as compared with a string stack is that it is possible to convert `c-addr n` strings into descriptor strings, or vice-versa, with a lightweight function call that does not modify or copy the string buffer. Finally, the descriptor based approach leaves the allocation and deallocation of string buffers to the application which is likely to be best placed to optimize resources based on its needs, rather than allocating an entire block of memory for a string stack at initialization.

However string stacks do have the advantage of being able to conveniently hold and manipulate multiple separate strings (for example to split or join). The descriptor approach could potentially assist in this case as noted below.

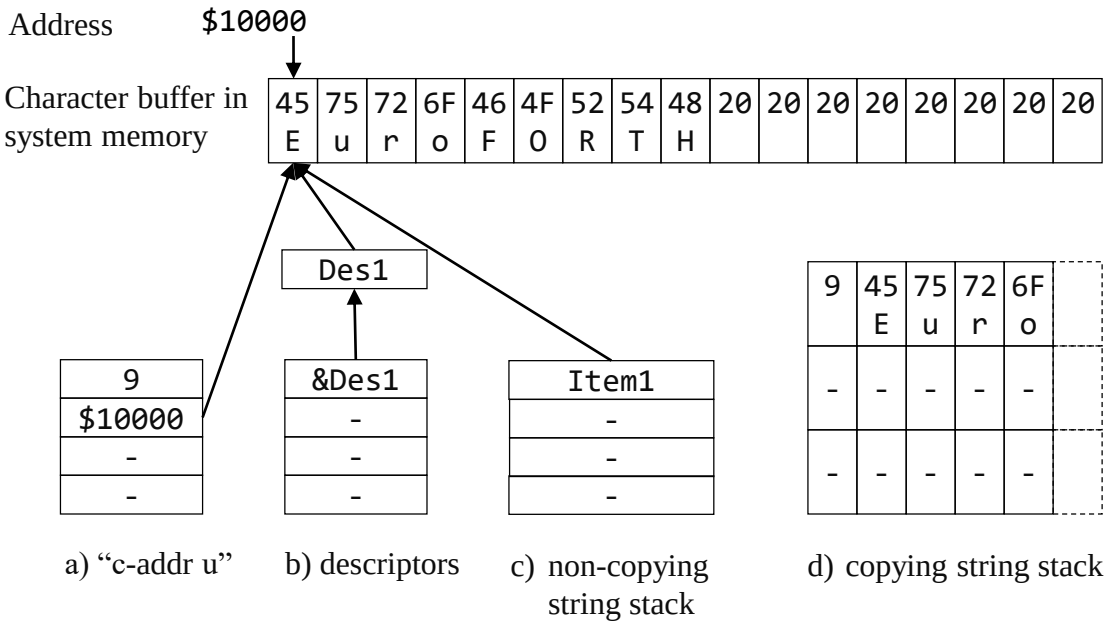

Figure 4: Comparison of alternative string representations. The "c-addr u" format holds the size and address of the string on the parameter stack. The descriptor approach holds the relevant string information in a descriptor that is pointed to by a single cell reference on the parameter stack. A non-copying string stack sets up a new stack to hold essentially the same information as is found in a string descriptor. In all of these three methods the character data itself is located in a separate character buffer that must be separately managed. Finally a copying string stack holds the length and character data directly on a separate stack and is responsible for managing the allocation and deallocation of that character data. Multiple string stack implementations exist and these examples are illustrative only.

We imagine that our descriptor-based approach could be extended to other abstract datatypes that have traditionally often been implemented on separate stacks. The most obvious candidates might be arbitrary-size integers or floating point numbers. Willi Striker has demonstrated that an efficient floating point arithmetic wordset can be obtained by holding floating point numbers as references on the parameter stack [9], and espoused as similar approach: "Everything on the stack is either an integer or a reference!" [10]. Another candidate datatype to benefit from a descriptor based approach might be collections, such as collections of strings themselves. This could assist in managing string intensive applications.

8.3 Response to Standardize Strings Now!

8.3.1 Memory management of character buffers

We have not solved the first issue that Anton raised: how to manage memory manage the character buffer. We explicitly leave allocation and deallocation of character buffer out of scope of our word-set. Our justification for this is that we want to implement our package in the Forth kernel at a foundational level and prefer to be adaptable to whatever kind of kernel-level character buffer any particular system adopts. These could vary widely.

We do provide an approach to the buffer allocation problems around concatenating strings. By separating the length of a string from the size of its buffer we allow for strings to be created ready with spare capacity. Whilst this approach still requires the programmer to anticipate the required capacity in advance, this appears to be a sensible compromise position between having no spare capacity at all (traditional Forth strings) and a some fully-dynamic but complicated approach. As pointed out in our introduction, this idea is not arbitrary but has the credential of being a core feature of array handling in Go and several other languages before it [1].

8.3.2 Convenient string representation

Our strings are conveniently held on the parameter stack as single cells, but do not suffer from any of the limitations of counted strings. Do the descriptors themselves create an inconvenience or risk of memory leaks? We argue not. Our system manages its own pool of descriptors and enforces automatic recycling when strings are consumed by string manipulation words, so the user has minimal work to do in remembering to $drop (rather than drop) strings that are no longer required. Errors in this regard can be minimized with stack comments and a brief code review.

8.4 Comparison with Go slices

A Go slice is a descriptor of an array segment. It consists of a pointer to the array, the length of the segment, and its capacity (the maximum length of the segment). In this, our descriptors are similar to Go slices. However Go slice functions may cause arrays to be reallocated, copied and otherwise memory managed. By contrast our words cannot, by design, affect the character buffers referenced by the descriptors. This is significantly less functionality, but consistent with our approach of leaving memory management elsewhere in the kernel.

8.5 Limitations

Naturally we acknowledge limitations introduced with our approach. The most significant limitation is the overhead introduced. Although our package is lightweight and performance overhead is therefore unlikely to be an issue, especially considering that I/O is the typical bottleneck in most string applications, extra memory is required and this must be reserved in advance for both the pool of descriptors and the string handling words themselves. Many small systems simply have no need for regular expressions or sophisticated string handling. It is enough to simply process the input steam and display messages to the user. In such cases the `c-addr u` format is both effective and efficient.

Another limitation is the fixed size of the descriptor pool. However with our stack-based approach to organizing free descriptors, the memory reserved for descriptors does not need to be contiguous and so it would not be too difficult to add further descriptors to the pool after initialization if to do so would be worthwhile.

9 Conclusion

We have presented a descriptor based approach to strings that draws on familiar ideas from other programming languages, but which we are not aware has been tried in Forth before. Although we have positioned this paper in part as a response to Anton Ertl's "Standardize Strings Now!", we do not necessarily suggest our approach as candidate for string standardization. Other approaches also have their merits, and the standard must weight common usage more highly than special innovations. As we also note above, our approach would also likely be a cost rather than a benefit to small systems.

However on systems where the string problem is relevant, we believe our approach offers an elegant and useful "middle-way" between the stark minimalism of `c-addr u` and the full-blown weight and complexity of string stacks with dynamic buffer management. Our intention is that our strings package be implemented at kernel level and used to support kernel-level string operations. To demonstrate the combination of easy and utility that our approach offers, we have developed a powerful yet lightweight descriptor-based regular expression matcher.

We acknowledge a debt to programming pioneer Rob Pike both for the exemplification of the slice mechanism in the Go Programming language and for his remarkably simple C regular expression matcher upon which our own is based. We wish to thank the anonymous academic reviewers for the helpful comments that greatly improved our paper.

References

[1] Go Slices: usage and internals, The Go Programming Language, January 2011, https://blog.golang.org/go-slices-usage-and-internals

[2] Sinclair ZX Spectrum BASIC Programming Manual, Chapter 8, 1982, Sinclair Research

[3] Informal Introduction to Algol 68, Section 2.5, Lindsey and van der Meulen, 2nd ed. 1977

[4] Fortran 77 Language Reference Manual, Chapter 2, Sun Microsystems, 2001

[5] Standardize Strings Now! Anton Ertl, TU Wien, EuroFORTH 2013

[6] REXX Parse (informal presentation), Carsten Strotmann, EuroFORTH 2013

[7] "Stack of Stacks", Ulrich Hoffmann, Forth Tagung 2017, https://wiki.forth-ev.de/doku.php/events:tagung-2017

[8] String Stack, Ulrich Hoffmann, first committed 2015, https://github.com/uho/stringstack

[9] Forth Floating Point Word-Set without Floating Point Stack, Willi Stricker, EuroFORTH 2013

[10] Video presentation (minutes 14:00 to 18:00), Willi Stricker, EuroForth 2013

[11] Article, http://www.cs.princeton.edu/courses/archive/spr09/cos333/beautiful.html, Rob Pike and Brian Kernighan, 1998

[12] RFC: Forth, a New Synthesis, Andrew Read and Ulrich Hoffmann, EuroForth 2017

[13] PatternForth: A Pattern-Matching Language for Real-Time Control, Brad Rodriguez, Bradley University, 1989