W65C816S STRING MANIPULATION LIBRARY V2

By BigDumbDinosaur

W65C816S STRING MANIPULATION LIBRARY V2 is a collection of 16-bit W65C816S native mode assembly language subroutines that perform a variety of useful character string operations. All functions are designed to take advantage of the W65C816S microprocessor's enhanced features and to facilitate integration with other native mode 65C816 software. No predefined storage allocation is required for any library function.

The following functions are implemented:

- `strcat` Catenate (concatenate) two strings.
- `strchr` Find a character in a string.
- `strcmp` Compare two strings.
- `strcpy` Copy a string to another string.
- `strdel` Delete a substring from a string.
- `strins` Insert a substring into a string.
- `strlen` Return the length of a string.
- `strpat` Compare two strings using wildcard pattern matching.
- `strprn` Print a character string.
- `strstr` Find a substring in a string.
- `strsub` Copy a substring from a string.

Please carefully read this document in its entirely before attempting to use any of this software.

DISTRIBUTION FILES

This distribution contains the following files:

- **`lib.str.65s`**

  `lib.str.65s` contains the assembly language source code that implements the various functions in the library. All source code was written and assembled in Michal Kowalski's 6502 simulator, version 1.2.12, with simulation set to the 65C02 mode. The binaries were tested on a W65C816S single-board computer running native mode firmware and are believed to be bug-free. Adaptation of the source code to other assemblers should not be too difficult.

  Parts or all of `lib.str.65s` must be `INCLUDE`d in any program that is to use string functions. As each function is a stand-alone subroutine, only the functions that are to be used need to be `INCLUDE`d. Other requirements may exist to utilize some functions. They will be mentioned as necessary.

- **`816macs.65s`**

  `816macs.65s` contains macroinstructions that synthesize W65C816S-specific instructions. The Kowalski assembler only knows the NMOS and Rockwell 65C02 instructions. Therefore, `816macs.65s` must be `INCLUDE`d in any program that is to use the string manipulation library, and must be `INCLUDE`d before any library function. Refer to comments in `816macs.65s` to see how these macros may be utilized in your own W65C816S native mode programs.

- **`stringmacs.65s`**

  `stringmacs.65s` contains macros that may be used in your programs to invoke the various functions supported by `lib.str.65s`, via a higher level syntax than what is required to directly call the library functions in pure assembly language. These macros use a syntax that resembles that of the comparable string manipulation functions found in ANSI C. Use is optional but is recommended in programs that make a lot of library calls.

GENERAL PROGRAMMING INFORMATION

In order to successfully integrate this string manipulation library in your programs you need to be aware of some programming requirements and considerations:

- This library uses the ANSI C convention of defining a character string—hereafter referred to as a string—as a sequence of non-null bytes that has been terminated by a null (`$00`) byte. For example, the string `ABCDE` would appear in a memory dump as `41 42 43 44 45 00`. A string may consist solely of a terminator, in which case it will be a null string.

- The maximum allowable string length is 32,767 characters plus the terminator. All functions will check string length and return an error if this length limit is violated.

- A string cannot span (overlap) memory banks. For example, the following string would not be acceptable to any library function:

  ```
  01FFF0 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52 00
  ```

  At byte value `$51`, the absolute address would be `$020000`, meaning the end of the string would be in bank `$02`, constituting bank overlap.

- The data bank defined in the microprocessor's `DB` (data bank) register is the bank in which string processing will occur. If necessary, your program should set a different bank before calling a library function. It is possible, though not trivial, to modify the library functions to accept data bank parameters, which would allow the string(s) being processed to be in alternate data banks.

- All functions expect 16-bit pointers to data, "data" being defined as a character, string or little-endian 16-bit integer value. In narrative text, a word such as `STRING1` refers to a string at the location pointed to by the address represented by the symbol `string1`. In macro and assembly language expositions, that `string1` is a pointer is implied.

- Pointers are pushed to the stack prior to the function call and must be of the correct number and in the correct order. At least one parameter is required for every call: a pointer to a string. Other pointers may be required for a second string and/or numeric data. The general assembly language form of a function call, as implemented in the macros, is:

  ```
  pea #ptr3†          ;3rd pointer, if required
  pea #ptr2           ;2nd pointer, if required
  pea #ptr1           ;string pointer (required)
  jsr function        ;call function
  bcs error           ;error occurred
  ```

†The description of `PEA` in the Eyes and Lichty 65C816 assembly language programming manual portrays the instruction as having an absolute addressing mode. Technically this is incorrect, as the microprocessor treats `PEA` as an immediate mode instruction—`PEA`'s data is the operand itself, and that data can be anything that will resolve to 16 bits. Therefore, `PEA` instructions will be shown as using immediate mode addressing in all programming examples to highlight this characteristic.

Note that parameters are pushed in the opposite order from that shown in the string macros—the macros will take care of this for you if you choose to use them. The called function will clear the parameters from the stack before returning to the calling program, eliminating the need for the calling function to do so.

WARNING! **Pushing an incorrect number of parameters or parameters of the wrong size prior to calling a function will cause the hardware stack to go out of balance when the function exits, possibly resulting in system fatality.**

Although the above function call example shows the use of `PEA` to push parameters (which is what the macros do), any instruction or sequence of instructions that can push a word (16-bit value) is acceptable, for example, `PEI` or even `PER`. Or, you can set a register to 16 bits, load it with the pointer and then push it. Use the method that is most convenient and efficient.

Most functions return with the accumulator and index registers as they were when the function was called. A few functions will return data in one or more registers. See the details for each function.

- All function calls must be tested for errors upon return. If an error occurs the function will return with the carry bit set in the microprocessor's status register (`SR`). Other `SR` bits may be manipulated to indicate the nature of the error. Error types may vary from function to function. If a function that normally returns data in one or more registers exits with an error, all registers will be as they were when the function was called. Refer to each function for details.

- The `strcat` and `strins` functions have the effect of expanding the target string (`STRING1`), which could lead to a buffer overflow if `STRING1` has not been allocated sufficient memory. Similarly, a buffer overflow could occur with the `strcpy` and `strsub` functions if the string being copied is longer than the space allocated to the target string. Library functions are not able to determine if a string will fit into a particular location—it is the calling function's responsibility to make that determination.

- All library functions use the 65C816's `MVN` and/or `MVP` instructions to copy data. Unlike other 65C816 instructions, `MVN` and `MVP` can be interrupted as they execute, which creates the potential for inadvertent disruption by an improperly designed interrupt service routine.

  As these instructions execute they maintain state information in all three registers. Therefore, *it is essential that the interrupt service routines on the system on which string library functions are to be run fully preserve the microprocessor state so that an interrupted* `MVN` *or* `MVP` *instruction can be restarted without error*. Succinctly stated, any register that is used by the interrupt service routine must be exactly restored prior to returning to the interrupted task.

WARNING! **Failure to properly preserve the microprocessor's state during interrupt processing could cause wild write operations to subsequently occur and depending on the design of the system, result in fatality.**

FUNCTION DESCRIPTIONS

In the following text, each function description will start with the macro call syntax for the function—macro names are all lower case, explain what the function does, describe any special processing cases that must be considered, and then finish with the assembly language call syntax used by the function. As previously noted, use of the macros in `stringmacs.65s`, which have the same names as the functions they invoke, is recommended for programing convenience and to assist in minimizing the likelihood of introducing bugs into your software.

- **`strcat string1,string2`**

  `strcat` copies `STRING2` to the end of `STRING1`, with the first character of `STRING2` overwriting the terminator of `STRING1`. Functionally, the operation can be characterized as `STRING1=STRING1+STRING2`. `STRING2` is not affected by this function unless it shares part of the address space of `STRING1`, in which case behavior may be undefined. If `string1` and `string2` point to the same string the result will be the same as if `A$=A$+A$` in BASIC were executed.

  Assembly language syntax:

```
      pea #s2ptr          ;STRING2's pointer
      pea #s1ptr          ;STRING1's pointer
      jsr strcat          ;catenate STRING2 to STRING1
      bcs error
      ---
error beq bankovr         ;bank overlap error
      bmi toolong         ;string too long error
```

```
Exit registers: .A: entry value
                .B: entry value
                .X: entry value
                .Y: entry value
```

  NOTE: This function uses self-modifying code.

- **`strchr string,char`**

  `strchr` scans `STRING` for the first occurrence of the character `CHAR` and if found, returns a pointer to `CHAR`'s position in `STRING`, as well as the number of instances of `CHAR` in `STRING`. Note that `char` does not point to a string; it points to a single character in memory.

  Assembly language syntax:

```
      pea #cptr           ;CHAR's pointer
      pea #sptr           ;STRING's pointer
      jsr strchr          ;scan STRING for CHAR
```

```
       bcs error
       beq notfound        ;CHAR not present in STRING
       ---
error  beq bankovr         ;bank overlap error
       bmi toolong         ;string too long

Exit registers: .A: entry value
                .B: entry value
                .X: 16-bit pointer to CHAR in STRING if found
                .Y: instances of CHAR in STRING
```

If `CHAR` is not present in `STRING`, `.X` and `.Y` will return `$0000`.

- **`strcmp string1,string2`**

`strcmp` compares `STRING2` to `STRING1` and returns the results via the microprocessor’s status register. Refer to the following table:

Z	N	Meaning
1	x	STRING2 = STRING1
0	0	STRING2 > STRING1
0	1	STRING2 < STRING1

In the above table, **`Z`** and **`N`** respectively refer to the zero and sign overflow bits in the status register. **<** means "less-than", **>** means "greater-than" and **`x`** means "don't care." Comparison relationships are based upon the binary values of the individual bytes in the strings. In the event there is a difference in string lengths the comparison result will be determined by the length of the shorter string. For example, if `STRING1="abcd123"` and `STRING2="abcd12"` then `STRING2 < STRING1`. Null strings are considered equal.

Assembly language syntax:

```
       pea #s2ptr          ;STRING2's pointer
       pea #s1ptr          ;STRING1's pointer
       jsr strcmp          ;compare STRING2 to STRING1
       bcs error
;
       beq equal           ;STRING2 = STRING1
       bmi lesser          ;STRING2 < STRING1
       bpl greater         ;STRING2 > STRING1
;
error  bmi toolong         ;string too long
       bpl bnkovr          ;bank overlap
```

```
Exit registers: .A: entry value
                .B: entry value
                .X: entry value
                .Y: entry value
```

- **`strcpy string1,string2`**

  `strcpy` copies `STRING2` to `STRING1`, overwriting the previous content of `STRING1`. An error will occur if `string1` and `string2` point to the same string. Behavior is undefined if `string2` points to an address within the bounds of `STRING1`.

  Assembly language syntax:

```
      pea #s2ptr          ;STRING2's pointer
      pea #s1ptr          ;STRING1's pointer
      jsr strcpy          ;copy STRING2 to STRING1
      bcs error
      ---
error beq bnkovr          ;bank overlap error
      bmi toolong         ;string too long error
      bvs s2_is_s1        ;string2 points to STRING1
```

```
Exit registers: .A: entry value
                .B: entry value
                .X: entry value
                .Y: entry value
```

  NOTE: This function uses self-modifying code.

- **`strdel string,i,n`**

  `strdel` deletes `N` characters from `STRING` starting at character `I` in `STRING`, contracting `STRING`'s length `N` characters. `I` is zero-based and must be less than the length of `STRING`. If the expression `I+N` equals or exceeds the length of `STRING`, `STRING` will be truncated to `I` characters. If `N=0` then `STRING` will not be modified—no error will occur in this case.

  Assembly language syntax:

```
      pea #nptr           ;N's pointer
      pea #iptr           ;I's pointer
      pea #sptr           ;STRING's pointer
      jsr strcpy          ;delete characters from STRING
      bcs error
      ---
error beq bnkovr          ;bank overlap error
      bmi toolong         ;string too long error
```

```
       bvs badindex        ;index exceeds STRING’s length
```

```
Exit registers: .A: entry value
                .B: entry value
                .X: entry value
                .Y: entry value
```

NOTE: This function uses self-modifying code.

- **`strins string1,string2,i`**

`strins` inserts `STRING2` into `STRING1` at character position `I`, causing `STRING1` to expand by the length of `STRING2`. `I` is zero-based and must be less than `STRING1`'s length. The combined length of `STRING1` and `STRING2` cannot exceed 32,767 characters.

Assembly language syntax:

```
       pea #iptr           ;I's pointer
       pea #s2ptr          ;STRING2's pointer
       pea #s1ptr          ;STRING1's pointer
       jsr strins          ;insert STRING2 into STRING
       bcs error
       ---
error  beq bnkovr          ;bank overlap error
       bmi toolong         ;string too long error
       bvs badindex        ;index exceeds STRING1’s length
```

```
Exit registers: .A: entry value
                .B: entry value
                .X: entry value
                .Y: entry value
```

NOTE: This function uses self-modifying code.

- **`strlen string`**

`strlen` returns the length of `STRING`, not including the terminator byte.

Assembly language syntax:

```
       pea #sptr           ;STRING's pointer
       jsr strlen          ;determine STRING's length
       bcs error
       ---
error  beq bnkovr          ;bank overlap error
       bmi toolong         ;string too long error
```

```
Exit registers: .C: 16 bits: STRING's length
                .X: entry value
                .Y: entry value
```

If there is no error and `STRING` is null, `$0000` will be returned in `.C`.

- **`strpat string,pattern`**

`strpat` compares `PATTERN` to `STRING` and returns the Boolean results `LIKE` or `UNLIKE`. The presence of the metacharacters `?` (character wildcard) and `*` (string wildcard) in `PATTERN` will affect the outcome of the comparison. Pattern matching behaves as follows:

? When present in `PATTERN`, a character wildcard will match exactly one character in `STRING`. For example, `fo?bar` will match `foobar`, `foObar`, `forbar`, etc. `????` will match `STRING` if `STRING` is exactly four characters in length, with the characters being anything, including `????`.

* When present in `PATTERN`, a string wildcard will match a sequence of characters in `STRING`. For example, `ab*` will match any `STRING` that begins with `ab`, such as `abcde` or `abracadabra`. `*yz` will match any `STRING` that ends with `yz`, such as `abcxyz`. `ab*lm*yz` will produce a match if `STRING` begins with `ab`, has `lm` anywhere in the middle, and ends with `yz`. For example, `ab*lm*yz` will match with the lower case Roman alphabet, as would `*lm*`.

Combinations of wildcards in `PATTERN` may be used in various ways. For example, `?*` will match any non-null `STRING`. Similarly, `???*` will match any `STRING` that is three or more characters in length, as would `*???`. The somewhat strange `PATTERN` `???*???` would match any `STRING` in which at least six characters are present.

The special case of `*` being the only character in `PATTERN` will match with any `STRING`, even if null. Multiple contiguous `*` characters in `PATTERN`, such as `***`, are treated as a single instance of `*`. Hence, `ab***yz` would be logically reduced to `ab*yz`.

If `PATTERN` has no wildcards `strpat` will act in a similar fashion to the `strcmp` function, but more slowly and without the “lesser-than” and “greater-than” results.

The comparison result is indicated by the microprocessor's `Z` flag as follows:

Z	Meaning
1	PATTERN ***LIKE*** STRING
0	PATTERN ***UNLIKE*** STRING

Behavior is undefined if `pattern` points to anywhere within `STRING`'s space..

Assembly language syntax:

```
         pea #pptr          ;PATTERN's pointer
         pea #sptr          ;STRING's pointer
         jsr strpat         ;compare PATTERN to STRING
         bcs error
         beq like           ;PATTERN LIKE STRING
         bne notequal       ;PATTERN UNLIKE STRING
         ---
error    bmi toolong        ;string too long
         bpl bnkovr         ;bank overlap
```

```
Exit registers: .A: entry value
                .B: entry value
                .X: entry value
                .Y: entry value
```

The complexity and length of `PATTERN` greatly affects `strpat`'s performance.

- **`strprn string`**

`strprn` prints `STRING` to an output device whose driver API entry point is defined by the symbol `putcha`. Output continues unabated until `STRING`'s terminator has been reached. The current version of `strprn` does not interpret anything in `STRING`. There are “hook points” in the code at which you may add features to perform interpretation, such as processing escape sequences or metacharacters. Refer to the source code for details.

`strprn` operates by making one subroutine call to `putcha` for each character in `STRING`, the character being made available to `putcha` in the eight bit accumulator. `putcha` is expected to block until it is able to process the character, and is also expected to preserve all registers, including `DB`. If these requirements are not met `strprn`'s behavior will be undefined. You may need to modify `strprn` to suit the requirements of your system's firmware or operating system.

Assembly language syntax:

```
         pea #sptr          ;STRING's pointer
         jsr strprn         ;print STRING
         bcs error
         ---
error    bmi toolong        ;string too long
         bpl bnkovr         ;bank overlap
```

```
Exit registers: .A: entry value
                .B: entry value
                .X: entry value
                .Y: entry value
```

- **strstr string1,string2**

  strstr returns a pointer to the first occurrence of STRING2 in STRING1 or a null pointer if STRING2 is not present in STRING1. Although it is possible to use strstr to locate a single character within STRING1 (STRING2 would be a single character plus a null), the strchr function is substantially faster in processing such a search and should be used for that purpose when practical.

  Assembly language syntax:

```
      pea #s2ptr         ;STRING2's pointer
      pea #s1ptr         ;STRING1's pointer
      jsr strstr         ;find STRING2 in STRING1
      bcs error
      beq notfound       ;STRING2 not in STRING1
      ---
error bmi toolong        ;string too long
      bpl bnkovr         ;bank overlap

Exit registers: .A: entry value
                .B: entry value
                .X: 16 bits: pointer to STRING2 in STRING1
                .Y: 16 bits: STRING2's length
```

- **strsub string1,string2,i,n**

  strsub copies N characters from STRING1 to STRING2, starting at index I, overwriting STRING2. I is zero-based and cannot be greater than or equal to the length of STRING1. If I+N is greater than or equal to STRING1's length all characters in STRING1, starting at I, will be copied to STRING2. If N=0 no copying will occur and STRING1 will not be changed in any way. Results are undefined if string2 points to anywhere within STRING1's space.

  Assembly language syntax:

```
      pea #nptr          ;N’s pointer
      pea #iptr          ;I’s pointer
      pea #s2ptr         ;STRING2's pointer
      pea #s1ptr         ;STRING1's pointer
      jsr strsub         ;copy substring from STRING1 to STRING2
      bcs error
      ---
error bvs badindex       ;index exceeds string length
      bmi toolong        ;string too long
      beq bnkovr         ;bank overlap
      bne s2_is_s1       ;string2 points to STRING1
```

```
Exit registers: .A: entry value
                .B: entry value
                .X: entry value
                .Y: entry value
```

NOTE: This function uses self-modifying code.