1.7: String handling library

These files have been made freely available by SGS-Thomson and may not be used to generate commercial products without explicit permission and agreed licensing terms OR placed in a public archive or given to third parties without explicit written permission from SGS-Thomson in Bristol.


Converted to HTML via RTFtoHTML, PERL5 and by hand by Dave Beckett <D.J.Beckett@ukc.ac.uk>.

WARNING: This has been converted from RTF format and may have mistakes compared to the printed version. If you find any in this file, please send them to me, and I will update it.

Dave Beckett <D.J.Beckett@ukc.ac.uk>

Library: string.lib

This library contains functions and procedures for handling strings and scanning lines of text. They assist with the manipulation of character strings such as names, commands, and keyboard responses. The library provides routines for:

- Identifying characters
- Comparing strings
- Searching strings
- Editing strings
- Scanning lines of text

<table>
<thead>
<tr>
<th>Result</th>
<th>Function</th>
<th>Parameter specifiers</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL</td>
<td>is.in.range</td>
<td>VAL BYTE char, bottom, top</td>
</tr>
<tr>
<td>BOOL</td>
<td>is.upper</td>
<td>VAL BYTE char</td>
</tr>
<tr>
<td>BOOL</td>
<td>is.lower</td>
<td>VAL BYTE char</td>
</tr>
<tr>
<td>BOOL</td>
<td>is.digit</td>
<td>VAL BYTE char</td>
</tr>
<tr>
<td>BOOL</td>
<td>is.hex.digit</td>
<td>VAL BYTE char</td>
</tr>
<tr>
<td>BOOL</td>
<td>is.id.char</td>
<td>VAL BYTE char</td>
</tr>
<tr>
<td>INT</td>
<td>compare.strings</td>
<td>VAL []BYTE str1, str2</td>
</tr>
<tr>
<td>BOOL</td>
<td>eqstr</td>
<td>VAL []BYTE s1, s2</td>
</tr>
<tr>
<td>INT</td>
<td>string.pos</td>
<td>VAL []BYTE search, str</td>
</tr>
<tr>
<td>INT</td>
<td>char.pos</td>
<td>VAL BYTE search</td>
</tr>
<tr>
<td>VAL []BYTE str</td>
<td>search.match</td>
<td>VAL []BYTE possibilities, str</td>
</tr>
<tr>
<td>INT, BYTE</td>
<td>search.no.match</td>
<td>VAL []BYTE possibilities, str</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Parameter Specifiers</th>
</tr>
</thead>
<tbody>
<tr>
<td>str.shift</td>
<td>[]BYTE str,</td>
</tr>
<tr>
<td></td>
<td>not.done</td>
</tr>
<tr>
<td>delete.string</td>
<td>INT len, []BYTE str,</td>
</tr>
<tr>
<td></td>
<td>not.done</td>
</tr>
<tr>
<td>insert.string</td>
<td>VAL []BYTE new.str,</td>
</tr>
<tr>
<td></td>
<td>not.done</td>
</tr>
<tr>
<td>to.upper.case</td>
<td>INT start, BOOL not.done</td>
</tr>
</tbody>
</table>

http://wotug.org/parallel/occam/documentation/inmos/string.html
1.7.1: Character identification

is.in.range

BOOL FUNCTION is.in.range (VAL BYTE char, bottom, top)

1. Returns TRUE if the value of char is in the range defined by bottom and top inclusive, otherwise returns FALSE.

is.upper

BOOL FUNCTION is.upper (VAL BYTE char)

1. Returns TRUE if char is an ASCII upper case letter, otherwise returns FALSE.

is.lower

BOOL FUNCTION is.lower (VAL BYTE char)

1. Returns TRUE if char is an ASCII lower case letter, otherwise returns FALSE.

is.digit

BOOL FUNCTION is.digit (VAL BYTE char)

1. Returns TRUE if char is an ASCII decimal digit, otherwise returns FALSE.

is.hex.digit

BOOL FUNCTION is.hex.digit (VAL BYTE char)

http://wotug.org/parallel/occam/documentation/inmos/string.html
1. Returns TRUE if char is an ASCII hexadecimal digit, otherwise returns FALSE. Upper or lower case letters A-F are allowed.

\( \text{is.id.char} \)

BOOL FUNCTION is.id.char (VAL BYTE char)

1. Returns TRUE if char is an ASCII character which can be part of an occam name; otherwise returns FALSE.

### 1.7.2: String comparison

1. These two procedures allow strings to be compared for order or for equality.

\( \text{compare.strings} \)

INT FUNCTION compare.strings (VAL []BYTE str1, str2)

1. This general purpose ordering function compares two strings according to the lexicographic ordering standard. (Lexicographic ordering is the ordering used in dictionaries etc., using the ASCII values of the bytes). It returns one of the 5 results 0, 1, -1, 2, or -2, as follows:

2. 0 The strings are exactly the same in length and content.
   1 str2 is a leading substring of str1
   -1 str1 is a leading substring of str2
   2 str1 is lexicographically later than str2
   -2 str2 is lexicographically later than str1
3. So if s is `abc`:

4. \( \text{compare.strings} ("abc", [s FROM 0 FOR 3]) = 0 \)
   \( \text{compare.strings} ("abc", [s FROM 0 FOR 2]) = 1 \)
   \( \text{compare.strings} ("abc", s) = -1 \)
   \( \text{compare.strings} ("bc", s) = 2 \)
   \( \text{compare.strings} ("a4", s) = -2 \)

\( \text{eqstr} \)

BOOL FUNCTION eqstr (VAL []BYTE s1, s2)

1. This is an optimized test for string equality. It returns TRUE if the two strings are the same size and have the same contents, FALSE otherwise.

### 1.7.3: String searching

1. These procedures allow a string to be searched for a match with a single byte or a string of bytes, for a byte which is one of a set of possible bytes, or for a byte which is not one of a set of bytes. Searches insensitive to alphabetic case should use to.upper.case or to.lower.case on both operands before using these procedures.

\( \text{string.pos} \)

INT FUNCTION string.pos (VAL []BYTE search, str)

1. Returns the position in str of the first occurrence of a substring which exactly matches search. Returns -1 if there is no such match.
INT FUNCTION char.pos (VAL BYTE search, VAL []BYTE str)

1. Returns the position in str of the first occurrence of the byte search. Returns -1 if there is no such byte.

search.match

INT, BYTE FUNCTION search.match (VAL []BYTE possibilities, str)

1. Searches str for any one of the bytes in the array possibilities. If one is found its index and identity are returned as results. If none is found then -1,255(BYTE) are returned.

search.no.match

INT, BYTE FUNCTION search.no.match (VAL []BYTE possibilities, str)

1. Searches str for a byte which does not match any one of the bytes in the array possibilities. If one is found its index and identity are returned as results. If none is found then -1,255(BYTE) are returned.

1.7.4: String editing

1. These procedures allow strings to be edited. The string to be edited is stored in an array which may contain unused space. The editing operations supported are: deletion of a number of characters and the closing of the gap created; insertion of a new string starting at any position within a string, which creates a gap of the necessary size.

2. These two operations are supported by a lower level procedure for shifting a consecutive substring left or right within the array. The lower level procedure does exhaustive tests against overflow.

str.shift

PROC str.shift ([]BYTE str, VAL INT start, len, shift, BOOL not.done)

1. Takes a substring [str FROM start FOR len], and copies it to a position shift places to the right. Any implied actions involving bytes outside the string are not performed and cause the error flag not.done to be set to TRUE. Negative values of shift cause leftward moves.

delete.string

PROC delete.string (INT len, []BYTE str, VAL INT start, size, BOOL not.done)

1. Deletes size bytes from the string str starting at str[start]. There are initially len significant characters in str and it is decremented appropriately. If start is outside the string, or start + size is greater than len, then no action occurs and not.done is set to TRUE.

insert.string

PROC insert.string (VAL []BYTE new.str, INT len, []BYTE str, VAL INT start, BOOL not.done)
1. Creates a gap in \texttt{str} starting at \texttt{str[start]} and copies the string \texttt{new.str} into it. There are initially \texttt{len} significant characters in \texttt{str} and \texttt{len} is incremented by the length of \texttt{new.str} inserted. Any overflow of the declared size of \texttt{str} results in truncation at the right and setting \texttt{not.done} to \texttt{TRUE}. This procedure may be used for simple concatenation on the right by setting \texttt{start = len} or on the left by setting \texttt{start = 0}. This method of concatenation differs from that using the \texttt{append} procedures in that it can never cause the program to stop.

\texttt{to.upper.case}

\begin{verbatim}
PROC to.upper.case ([][BYTE str)
1. Converts all alphabetic characters in \texttt{str} to upper case. All other characters are left unaltered.
\end{verbatim}

\texttt{to.lower.case}

\begin{verbatim}
PROC to.lower.case ([][BYTE str)
1. Converts all alphabetic characters in \texttt{str} to lower case. All other characters are left unaltered.
\end{verbatim}

\texttt{append.char}

\begin{verbatim}
PROC append.char (INT len, [][]BYTE str,
VAL BYTE char)
1. Writes a byte \texttt{char} into the array \texttt{str} at \texttt{str[len]}. \texttt{len} is incremented by 1. Behaves like \texttt{STOP} if the array overflows.
\end{verbatim}

\texttt{append.text}

\begin{verbatim}
PROC append.text (INT len, [][]BYTE str,
VAL [][]BYTE text)
1. Writes a string \texttt{text} into the array \texttt{str}, starting at \texttt{str[len]} and computing a new value for \texttt{len}. Behaves like \texttt{STOP} if the array overflows.
\end{verbatim}

\texttt{append.int}

\begin{verbatim}
PROC append.int (INT len, [][]BYTE str,
VAL INT number, width)
1. Converts \texttt{number} into a sequence of ASCII decimal digits padded out with leading spaces and an optional sign to the specified field width, \texttt{width}, if necessary. If the number cannot be represented in \texttt{width} characters it is widened as necessary. A zero value for \texttt{width} will give minimum width. The converted number is written into the array \texttt{str} starting at \texttt{str[len]} and \texttt{len} is incremented. Behaves like \texttt{STOP} if the array overflows or if \texttt{width < 0}.
\end{verbatim}

\texttt{append.int64}

\begin{verbatim}
PROC append.int64 (INT len, [][]BYTE str,
VAL INT64 number,
VAL INT width)
1. As \texttt{append.int} but for 64-bit integers.
\end{verbatim}

\texttt{append.hex.int}

\begin{verbatim}
PROC append.hex.int (INT len, [][]BYTE str,
VAL INT number, width)
\end{verbatim}
1. Converts number into a sequence of ASCII hexadecimal digits, using upper case letters, preceded by `#'. The total number of characters set is always width+1, padding out with `0' or `F' on the left if necessary. The number is truncated at the left if the field is too narrow, thereby allowing the less significant part of any number to be printed. The converted number is written into the array str starting at str[len] and len is incremented. Behaves like STOP if the array overflows or if width < 0.

append.hex.int64

PROC append.hex.int64 (INT len, [BYTE str,
VAL INT64 number,
VAL INT width)

1. As append.hex.int but for 64-bit integers.

append.real32

PROC append.real32 (INT len, [BYTE str,
VAL REAL32 number,
VAL INT Ip, Dp)

1. Converts number into a sequence of ASCII characters formatted using Ip and Dp as described under REAL32TOSTRING (see section 1.8).
2. The converted number is written into the array str starting at str[len] and len is incremented. Behaves like STOP if the array overflows.

append.real64

PROC append.real64 (INT len, [BYTE str,
VAL REAL64 number,
VAL INT Ip, Dp)

1. As append.real32, but for 64-bit real values. The formatting variables Ip and Dp are described under REAL32TOSTRING (see section 1.8).

1.7.5: Line parsing

1. Depending on the initial value of the variable ok these two procedures either read a line serially, returning the next word and next integer respectively, or the procedures act almost like a SKIP (see below). The user should initialize the variable ok as appropriate.

next.word.from.line

PROC next.word.from.line (VAL [BYTE line,
INT ptr, len,
[BYTE word,
BOOL ok)

1. If ok is passed in as TRUE, on entry to the procedure, skips leading spaces and horizontal tabs and reads the next word from the string line. The value of ptr is the starting point of the search. A word continues until a space or tab or the end of the string line is encountered. If the end of the string is reached without finding a word, the boolean ok is set to FALSE, and len is 0. If a word is found but is too large for word, then ok is set to FALSE, but len will be the length of the word that was found; otherwise the found word will be in the first len bytes of word. The index ptr is updated to be that of the space or tab immediately after the found word, or is SIZE line. If ok is passed in as FALSE, len is set to 0, ptr and ok remain unchanged, and word is undefined.
next.int.from.line

PROC next.int.from.line (VAL []BYTE line,
INT ptr, number,
BOOL ok)

1. If ok is passed in as TRUE, on entry to the procedure, skips leading spaces and horizontal tabs and reads the next integer from the string line. The value of ptr is the starting point of the search. The integer is considered to start with the first non-space, non-tab character found and continues until a space or tab or the end of the string line is encountered. If the first sequence of non-space, non-tab characters does not exist, does not form an integer, or forms an integer that overflows the INT range then ok is set to FALSE, and number is undefined; otherwise ok remains TRUE, and number is the integer read. A `+' or `-` may be the first character of the integer. The index ptr is updated to be that of the space or tab immediately after the found integer, or is SIZE line. If ok is passed in as FALSE, then ptr and ok remain unchanged, and number is undefined.