International Standards Organization

Varying Length Character Strings

in

Fortran

ISO/IEC 1539-2
{auxiliary standard to ISO/IEC 1539 : 1991}

{Second Committee Draft Produced 26-Jul-93}
Contents

Foreword ................................................................. v

Introduction ........................................................... vi

Section 1 : Scope ....................................................... 1
  1.1 Normative References ......................................... 2

Section 2 : Requirements ............................................... 3
  2.1 The Name of the Module ....................................... 3
  2.2 The Type ....................................................... 3
  2.3 Extended Meanings for Intrinsic Operators ..................... 3
    2.3.1 Assignment ............................................. 3
    2.3.2 Concatenation ......................................... 4
    2.3.3 Comparisons .......................................... 4
  2.4 Extended Meanings for Generic Intrinsic Procedures ............. 4
    2.4.1 The LEN Procedure .................................... 4
    2.4.2 The CHAR Procedure .................................. 5
    2.4.3 The ICHAR Procedure .................................. 5
    2.4.4 The IACHAR Procedure .................................. 5
    2.4.5 The TRIM procedure .................................... 5
    2.4.6 The LEN_TRIM procedure .............................. 6
    2.4.7 The ADJUSTL procedure ............................... 6
    2.4.8 The ADJUSTR procedure ............................... 6
    2.4.9 The REPEAT procedure ................................ 7
    2.4.10 Comparison Procedures .............................. 7
    2.4.11 The INDEX procedure ................................ 8
    2.4.12 The SCAN procedure ................................. 8
    2.4.13 The VERIFY procedure .............................. 9
  2.5 Additional Generic Procedure for Type Conversion .............. 9
    2.5.1 The VAR_STR procedure ............................... 9
  2.6 Additional Generic Procedures for Input/Output ............... 9
    2.6.1 The GET procedure .................................... 10
    2.6.2 The PUT procedure .................................... 11
    2.6.3 The PUT_LINE procedure ............................. 11
  2.7 Additional Generic Procedures for Substring Manipulation .... 12
    2.7.1 The INSERT procedure ............................... 12
    2.7.2 The REPLACE procedure .............................. 12
    2.7.3 The REMOVE procedure ................................ 13
    2.7.4 The EXTRACT procedure .............................. 14
    2.7.5 The SPLIT procedure ................................ 14

Annex A ............................................................... 15
  MODULE ISO_VARYING_STRING .................................. 15
Annex B

PROGRAM word_count
PROGRAM vocabulary_word_count
Foreword

[This page to be provided by ISO CS]
Introduction

This International Standard has been prepared by ISO/IEC JTC1/SC22/WG5, the technical working group for the Fortran language. This International Standard is an auxiliary standard to ISO/IEC 1539 : 1991, which defines the latest revision of the Fortran language, and is the first part of the multipart Fortran family of standards; this International Standard is the second part. The revised language defined by the above standard is informally known as Fortran 90.

This International Standard defines the interface and semantics for a module which provides facilities for the manipulation of character strings of arbitrary and dynamically variable length. The annex A includes a possible implementation in Fortran 90 of a module that conforms to this International Standard. It should be noted, however, that this is purely for purposes of demonstrating the feasibility and portability of the standard. The actual code shown in this annex is not intended in any way to prescribe the method of implementation, nor is there any implication that this is in any way an optimal portable implementation. The module is merely a fairly straightforward demonstration that a portable implementation is possible.
Section 1 : Scope

This International Standard defines facilities for use in Fortran for the manipulation of character strings of dynamically variable length. This International Standard provides an auxiliary standard for the Fortran language informally known as Fortran 90. The standard defining this revision of the Fortran language is

- ISO/IEC 1539 : 1991 "Programming Language Fortran"

This International Standard is an auxiliary standard to that defining Fortran 90 in that it defines additional facilities to those defined intrinsically in the primary language standard. However, a processor conforming to the Fortran 90 standard is not required to also conform to this International Standard. Nevertheless, conformance to to this International Standard assumes conformance to the primary Fortran 90 standard.

This International Standard prescribes the name of a Fortran module, the name of the derived data type to be used to represent varying-length strings, the interfaces for the procedures and operators that must be provided to manipulate objects of this type, and the semantics that are required for each of the entities made accessible by this module.

This International Standard does not prescribe the details of any implementation. Neither the method used to represent the data entities of the defined type nor the algorithms used to implement the procedures or operators whose interfaces are defined by this International Standard are prescribed. A conformant implementation may use any representation and any algorithms, subject only to the requirement that the publicly accessible names and interfaces conform to this International Standard, and that the semantics are as required by this International Standard and those of ISO/IEC 1539 : 1991.

It should be noted that a processor is not required to implement this International Standard in order to be a standard conforming Fortran processor, but if a processor implements facilities for manipulating varying length character strings, it is recommended that this be done in a manner that is conformant with this International Standard. A processor conforming to this International Standard may extend the facilities provided for the manipulation of varying length character strings as long as such extensions do not conflict with those defined in this International Standard.

A module, written in standard conforming Fortran, is included in Annex A. This module illustrates one way in which a standard conforming module could be written. This module is both conformant with the requirements of this International Standard and, because it is written in standard conforming Fortran, it provides a portable implementation of the required facilities.

This module is included for information only and is not intended to constrain implementations in any way. This module is a demonstration that at least one implementation, in standard conforming and hence portable Fortran, is possible.

It should be noted that this International Standard defines facilities for dynamically varying length strings of characters of default kind only. Throughout this International Standard all references to intrinsic type CHARACTER should be read as meaning characters of default kind. Similar facilities could be defined for non-default kind characters by a separate, if similar, module for each such character kind.
This International Standard has been designed, as far as is reasonable, to provide for varying length character strings the facilities that are available for intrinsic fixed length character strings. All the intrinsic operations and functions which apply to fixed length character strings have extended meanings defined by this International Standard for varying length character strings. Also a small number of additional facilities are defined that are appropriate because of the essential differences between the intrinsic type and the varying length derived data type.

1.1 Normative References

- ISO/IEC 1539 : 1991 "Programming Language Fortran"
- ISO/IEC 646 : 1983 "Character Coding"
Section 2 : Requirements

2.1 The Name of the Module

The name of the module shall be

\texttt{ISO\_VARYING\_STRING}

Programs shall be able to access the facilities defined by this International Standard by the inclusion of \texttt{USE} statements of the form

\texttt{USE ISO\_VARYING\_STRING}

2.2 The Type

The type shall have the name

\texttt{VARYING\_STRING}

Entities of this type shall represent values which are strings of characters of default kind. These character strings may be of any non-negative length and this length may vary dynamically during the execution of a program. There shall be no arbitrary upper length limit other than that imposed by the size of the processor and the complexity of the programs it is able to process. The characters representing the value of the string have positions 1,2,...,N, where N is the length of the string. The internal structure of the type shall be \texttt{PRIVATE} to the module.

2.3 Extended Meanings for Intrinsic Operators

The meanings for the intrinsic operators of:

\begin{itemize}
  \item \texttt{assignment} =
  \item \texttt{concatenation} //
  \item \texttt{comparisons} ==, /=, <, <=, >=, >
\end{itemize}

shall be extended to accept any combination of scalar operands of type \texttt{VARYING\_STRING} and type \texttt{CHARACTER}. Note that, the equivalent comparison operator forms, \texttt{.EQ.}, \texttt{.NE.}, \texttt{.LT.}, \texttt{.LE.}, \texttt{.GE.}, \texttt{.GT.}, also have their meanings extended in this manner.

2.3.1 Assignment: An assignment of the form

\texttt{var = expr}

shall be defined for scalars with the following type combinations:

\begin{itemize}
  \item \texttt{VARYING\_STRING = VARYING\_STRING}
  \item \texttt{VARYING\_STRING = CHARACTER}
  \item \texttt{CHARACTER = VARYING\_STRING}
\end{itemize}

Action: The characters that are the value of the expression \texttt{expr} become the value of the variable \texttt{var}. There are two cases:

\begin{itemize}
  \item Case(i) : Where the variable is of type \texttt{VARYING\_STRING}, the length of the variable becomes that of the expression.
  \item Case(ii) : Where the variable is of type \texttt{CHARACTER}, the rules of intrinsic assignment to a Fortran character variable apply. Namely, if the expression string is longer than the declared length of the character variable, only the left-most characters are assigned. If the character variable is longer than that of the string expression, it is padded on the right with blanks.
\end{itemize}
2.3.2 Concatenation: The concatenation operation

\[ \text{string}_a \ // \ \text{string}_b \]

shall be defined for scalars with the following type combinations:

- \text{VARYING\_STRING} // \text{VARYING\_STRING}
- \text{VARYING\_STRING} // \text{CHARACTER}
- \text{CHARACTER} // \text{VARYING\_STRING}

The values of the operands are unchanged by the operation.

Result Type: \text{VARYING\_STRING}

Result Value: The result value is a new string whose characters are the same as those produced by concatenating the operand character strings in the order given.

2.3.3 Comparisons: Comparisons of the form

\[ \text{string}_a \ .\text{OP.} \ \text{string}_b \]

where \text{.OP.} represents any of the operators ==, /=, <, <=, >=, > shall be defined for scalar operands with the following type combinations:

- \text{VARYING\_STRING} .\text{OP.} \text{VARYING\_STRING}
- \text{VARYING\_STRING} .\text{OP.} \text{CHARACTER}
- \text{CHARACTER} .\text{OP.} \text{VARYING\_STRING}

The values of the operands are unchanged by the operation.

Note that, the equivalent operator forms .EQ., .NE., .LT., .LE., .GE., .GT. also have their meanings extended in this manner.

Result Type: default \text{LOGICAL}

Result Value: The result value is true if \text{string}_a stands in the indicated relation to \text{string}_b. The collating sequence used for the inequality comparisons is that defined by the processor for characters of default kind. If \text{string}_a and \text{string}_b are of different lengths, the comparison is done as if the shorter string were padded on the right with blanks.

2.4 Extended Meanings for Generic Intrinsic Procedures

The generic intrinsic procedures \text{LEN, CHAR, ICHAR, IACHAR, TRIM, LEN\_TRIM, ADJUSTL, ADJUSTR, REPEAT, LLT, LLE, LGE, LGT, INDEX, SCAN, and VERIFY} shall have their meanings extended to include the appropriate scalar argument type combinations involving \text{VARYING\_STRING} and \text{CHARACTER}.

2.4.1 The LEN Procedure: The generic function reference of the form

\[ \text{LEN}(\text{string}) \]

shall be added.

Description: Returns the length of a character string.

Class: Transformational function.

Argument: \text{string} is a scalar of type \text{VARYING\_STRING}. The argument is unchanged by the procedure.

Result Type: default \text{INTEGER}

Result Value: The result value is the number of characters in \text{string}.
2.4.2 The CHAR Procedure: The generic function references of the form

\[
\text{CHAR}(\text{string}) \\
\text{CHAR}(\text{string}, \text{length})
\]

shall be added.

**Description:** Converts a varying string value to default character.

**Class:** Transformational function.

**Arguments:**

- \text{string} - is of type \text{VARYING\_STRING}
- \text{length} - is of type default \text{INTEGER}.

The arguments must be scalars and are unchanged by the procedure.

**Result Type:** default \text{CHARACTER}.

**Result Value:**

1. If \text{length} is absent, the result has the value of the characters of \text{string}, and the same length.
2. If \text{length} is present, the result has the length specified by the argument \text{length}. If \text{string} is longer than \text{length}, the result is truncated on the right. If \text{string} is shorter than \text{length}, the result is padded on the right with blanks. If \text{length} is less than one, the result is of zero length.

2.4.3 The ICHAR Procedure: The generic function reference of the form

\[
\text{ICHAR}(c)
\]

shall be added.

**Description:** Position of a character in the processor collating sequence.

**Class:** Transformational function.

**Argument:** \(c\) is a scalar of type \text{VARYING\_STRING} and of length exactly one. The argument is unchanged by the procedure.

**Result Type:** default \text{INTEGER}.

**Result Value:** The result value is the position of the character \(c\) in the processor defined collating sequence for default characters.

2.4.4 The IACHAR Procedure: The generic function reference of the form

\[
\text{IACHAR}(c)
\]

shall be added.

**Description:** Position of a character in the collating sequence defined by the standard ISO 646 : 1983.

**Class:** Transformational function.

**Argument:** \(c\) is a scalar of type \text{VARYING\_STRING} and of length exactly one. The argument is unchanged by the procedure.

**Result Type:** default \text{INTEGER}.

**Result Value:** The result value is the position of the character \(c\) in the collating sequence defined by the standard ISO 646 : 1983 for default characters. If the character \(c\) is not defined in the standard set, the result is processor dependent.

2.4.5 The TRIM procedure: The generic function reference of the form

\[
\text{TRIM}(\text{string})
\]

shall be added.

**Description:** Remove trailing blanks from a string.

**Class:** Transformational Function.

**Argument:** \text{string} is a scalar of type \text{VARYING\_STRING}. The argument is unchanged by the procedure.
Result Type: \texttt{VARYING\_STRING}.

Result Value: The result value is the string produced by removing any trailing blanks from the argument. If the argument \texttt{string} contains only blank characters or is of zero length, the result is a zero-length string.

\subsection*{2.4.6 The \texttt{LEN\_TRIM} procedure}

The generic function reference of the form

\begin{verbatim}
LEN\_TRIM(string)
\end{verbatim}

shall be added.

Description: Length of a string not counting any trailing blanks.

Class: Transformational function.

Argument: \texttt{string} is a scalar of type \texttt{VARYING\_STRING}. The argument is unchanged by the procedure.

Result Type: default \texttt{INTEGER}.

Result Value: The result value is the position of the last non-blank character in \texttt{string}. If the argument \texttt{string} contains only blank characters or is of zero length, the result is zero.

\subsection*{2.4.7 The \texttt{ADJUSTL} procedure}

The generic function reference of the form

\begin{verbatim}
ADJUSTL(string)
\end{verbatim}

shall be added.

Description: Adjusts to the left, removing any leading blanks and inserting trailing blanks.

Class: Transformational function.

Argument: \texttt{string} is a scalar of type \texttt{VARYING\_STRING}. The argument is unchanged by the procedure.

Result Type: \texttt{VARYING\_STRING}.

Result Value: The result value contains the same characters as the argument shifted cyclically to the left until the first character is non-blank. The result is identical to the argument if the first character of \texttt{string} is non-blank, \texttt{string} contains only blank characters or is of zero length.

\subsection*{2.4.8 The \texttt{ADJUSTR} procedure}

The generic function reference of the form

\begin{verbatim}
ADJUSTR(string)
\end{verbatim}

shall be added.

Description: Adjusts to the right, removing any trailing blanks and inserting leading blanks.

Class: Transformational function.

Argument: \texttt{string} is a scalar of type \texttt{VARYING\_STRING}. The argument is unchanged by the procedure.

Result Type: \texttt{VARYING\_STRING}.

Result Value: The result value contains the same characters as the argument shifted cyclically to the right until the last character is non-blank. The result is identical to the argument if the last character of \texttt{string} is non-blank, \texttt{string} contains only blank characters or is of zero length.
2.4.9 The REPEAT procedure: The generic function reference of the form

\[ \text{REPEAT}(\text{string}, \text{ncopies}) \]

shall be added.

**Description:** Concatenate several copies of a string.

**Class:** Transformational function.

**Arguments:**
- `string` - is a scalar of type `VARYING_STRING`,
- `ncopies` - is a scalar of type default `INTEGER`.

The value of `ncopies` must not be negative. The arguments are unchanged by the procedure.

**Result Type:** `VARYING_STRING`.

**Result Value:** The result value is the string produced by repeated concatenation of the argument `string`, producing a string containing `ncopies` copies of `string`. A negative value for `ncopies` is not permitted. If `ncopies` is zero, the result is of zero length.

2.4.10 Comparison Procedures: The set of generic function references of the form

\[ \text{Lop}(\text{string}_a, \text{string}_b) \]

shall be added, where `op` stands for one of:
- `LT` - less than
- `LE` - less than or equal to
- `GE` - greater than or equal to
- `GT` - greater than

**Description:** Compares the lexical ordering of two strings based on the ISO 646 : 1983 collating sequence.

**Class:** Transformational function.

**Arguments:** `string_a` and `string_b` are scalars of one of the type combinations:
- `VARYING_STRING` and `VARYING_STRING`,
- `VARYING_STRING` and `CHARACTER`, or
- `CHARACTER` and `VARYING_STRING`.

The arguments are unchanged by the procedure.

**Result Type:** default `LOGICAL`.

**Result Value:** The result value is true if `string_a` stands in the indicated relationship to `string_b`, and is false otherwise. The collating sequence used to establish the ordering of characters for these procedures is that of the International Standard, ISO 646 : 1983. If `string_a` and `string_b` are of different length, the comparison is done as if the shorter string were padded on the right with blanks. If either argument contains a character not defined by the standard, the result value is processor dependent.
2.4.11 The INDEX procedure: The generic function reference of the form

\[
\text{INDEX}(\text{string}, \text{substring}, \text{back})
\]

shall be added.

Description: Returns an integer which is the starting position of a substring within a string.

Class: Transformational function.

Arguments. string and substring are scalars of one of the type combinations:

- VARYING_STRING and VARYING_STRING,
- CHARACTER and VARYING_STRING, or
- VARYING_STRING and CHARACTER.

back - is a scalar of type default LOGICAL and is OPTIONAL.

Result type: default INTEGER.

Result value:

Case(i) : If back is absent or is present with the value false, the result is the minimum positive value of I such that,

\[
\text{EXTRACT}(\text{string}, I, I + \text{LEN}(\text{substring}) - 1) = \text{substring}
\]

or zero if there is no such value. Zero is returned if \(\text{LEN}(\text{string}) < \text{LEN}(\text{substring})\), and one is returned if \(\text{LEN}(\text{substring}) = 0\).

Case(ii) : If back is present with the value true and if string contains at least one character that is in set, the value of the result is the position of the rightmost character of string that is in set.

Case(iii) : The value of the result is zero if no character of string is in set or if the length of either string or set is zero.

2.4.12 The SCAN procedure: The generic function reference of the form

\[
\text{SCAN}(\text{string}, \text{set}, \text{back})
\]

shall be added.

Description: Scan a string for any one of the characters in a set of characters.

Class: Transformational function.

Arguments: string and set are scalars of one of the type combinations:

- VARYING_STRING and VARYING_STRING,
- VARYING_STRING and CHARACTER, or
- CHARACTER and VARYING_STRING.

back - is a scalar of type default LOGICAL and is OPTIONAL.

The arguments are unchanged by the procedure.

Result Type: default INTEGER.

Result Value:

Case(i) : If back is absent or is present with the value false and if string contains at least one character that is in set, the value of the result is the position of the leftmost character of string that is in set.

Case(ii) : If back is present with the value true and if string contains at least one character that is in set, the value of the result is the position of the rightmost character of string that is in set.

Case(iii) : The value of the result is zero if no character of string is in set or if the length of either string or set is zero.
2.4.13 The VERIFY procedure: The generic function reference of the form

\[
\text{VERIFY(string, set, back)}
\]

shall be added.

**Description:** Verify that a string contains only characters from a given set by scanning for any character not in the set.

**Class:** Transformational function.

**Arguments:** `string` and `set` are scalars of one of the type combinations:
- `VARYING_STRING` and `VARYING_STRING`,
- `VARYING_STRING` and `CHARACTER`, or
- `CHARACTER` and `VARYING_STRING`.

`back` - is a scalar of type default LOGICAL and is **OPTIONAL**. The arguments are unchanged by the procedure.

**Result Type:** default **INTEGER**.

**Result Value:**
- Case(i) : If `back` is absent or is present with the value false and if `string` contains at least one character that is not in `set`, the value of the result is the position of the leftmost character of `string` that is not in `set`.
- Case(ii) : If `back` is present with the value true and if `string` contains at least one character that is not in `set`, the value of the result is the position of the rightmost character of `string` that is not in `set`.
- Case(iii) : The value of the result is zero if each character of `string` is in `set` or if the length of `string` is zero.

#### 2.5 Additional Generic Procedure for Type Conversion

An additional generic procedure shall be added to convert scalar intrinsic fixed-length character values into scalar varying-length string values.

2.5.1 The VAR_STR procedure: The generic function reference of the form

\[
\text{VAR_STR(char)}
\]

shall be provided.

**Description:** Converts an intrinsic fixed-length character value into the equivalent varying-length string value.

**Class:** Transformational function.

**Argument:** `char` is a scalar of type default `CHARACTER` and may be of any length. The argument is unchanged by the procedure.

**Result Type:** `VARYING_STRING`.

**Result Value:** The result value is the same string of characters as the argument.

#### 2.6 Additional Generic Procedures for Input/Output

The following additional generic procedures shall be provided to support input and output of varying-length string values with formatted sequential files.

- `GET` - input part or all of a record into a string
- `PUT` - append a string to an output record
- `PUT_LINE` - append a string to an output record and end the record
2.6.1 The GET procedure: The generic subroutine references of the forms

```fortran
CALL GET(string,maxlen,iostat)
CALL GET(unit,string,maxlen,iostat)
CALL GET(string,set,maxlen,iostat)
CALL GET(unit,string,set,maxlen,iostat)
```

shall be provided.

**Description:** Input characters from an external file into a string.

**Class:** Subroutine.

**Arguments:**

- `string` - is of type `VARYING_STRING`.
- `maxlen` - is of type default `INTEGER` and is `OPTIONAL`.
- `unit` - is of type default `INTEGER`.
- `set` - is either of type `VARYING_STRING` or of type `CHARACTER`
- `iostat` - is of type default `INTEGER` and is `OPTIONAL`.

All arguments are scalar. The argument `unit` specifies the input unit to be used. It must be connected to a formatted file for sequential read access. If the argument `unit` is omitted, the default input unit is used.

**Action:** The GET procedure causes characters from the connected file, starting with the next character in the current record if there is a current record or the first character of the next record if not, to be read and stored in the variable `string`. The end of record always terminates the input but input may be terminated before this. If `maxlen` is present, its value indicates the maximum number of characters that will be read. If `maxlen` is less than or equal to zero, no characters will be read and `string` will be set to zero length. If `maxlen` is absent, a maximum of `HUGE(1)` is used. If the argument `set` is provided, this specifies a set of characters the occurrence of any of which will terminate the input. This terminal character, although read from the input file, will not be included in the result string. The file position after the data transfer is complete is after the last character that was read. If the transfer was terminated by the end of record being reached, the file is positioned after the record just read. If present, the argument `iostat` is used to return the status resulting from the data transfer. A zero value is returned if a valid read operation occurs, a positive value if an error is caused, and a negative value if an end-of-file condition occurs. If `iostat` is absent and anything other than a valid read operation occurs, the program execution is terminated.
2.6.2 The PUT procedure: The generic subroutine references of the forms

```fortran
CALL PUT(string,iostat)
CALL PUT(unit,string,iostat)
```

shall be provided.

**Description:** Output a string to an external file.

**Class:** Subroutine.

**Argument:**

- `string` - is either type `VARYING_STRING` or type `CHARACTER`.
- `unit` - is type default `INTEGER`.
- `iostat` - is type default `INTEGER` and is `OPTIONAL`.

All arguments are scalar. The argument `unit` specifies the output unit to be used. It must be connected to a formatted file for sequential write access. If the argument `unit` is omitted, the default output unit is used.

**Action:** The PUT procedure causes the characters of `string` to be appended to the current record, if there is a current record, or to the start of the next record if there is no current record. The last character transferred becomes the last character of the current record, which is the last record of the file. If present, the argument `iostat` is used to return the status resulting from the data transfer. A zero value is returned if a valid write operation occurs, and a positive value if an error occurs. If `iostat` is absent and anything other than a valid write operation occurs, the program execution is terminated.

2.6.3 The PUT_LINE procedure: The generic subroutine references of the forms

```fortran
CALL PUT_LINE(string,iostat)
CALL PUT_LINE(unit,string,iostat)
```

shall be provided.

**Description:** Output a string to an external file and end the record.

**Class:** Subroutine.

**Argument:**

- `string` - is either type `VARYING_STRING` or type `CHARACTER`
- `unit` - is type default `INTEGER`
- `iostat` - is type default `INTEGER` and is `OPTIONAL`.

All arguments are scalar. The argument `unit` specifies the output unit to be used. It must be connected to a formatted file for sequential write access. If the argument `unit` is omitted, the default output unit is used.

**Action:** The PUT_LINE procedure causes the characters of `string` to be appended to the current record, if there is a current record, or to the start of the next record if there is no current record. Following completion of the data transfer, the file is positioned after the record just written, which becomes the previous and last record of the file. If present, the argument `iostat` is used to return the status resulting from the data transfer. A zero value is returned if a valid write operation occurs, and a positive value if an error occurs. If `iostat` is absent and anything other than a valid write operation occurs, the program execution is terminated.
2.7 Additional Generic Procedures for Substring Manipulation

The following additional generic procedures shall be provided to support the manipulation of scalar substrings of scalar varying-length strings.

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>INSERT</strong></td>
<td>insert a substring into a string</td>
</tr>
<tr>
<td><strong>REPLACE</strong></td>
<td>replace a substring in a string</td>
</tr>
<tr>
<td><strong>REMOVE</strong></td>
<td>remove a section of a string</td>
</tr>
<tr>
<td><strong>EXTRACT</strong></td>
<td>extract a section from a string</td>
</tr>
<tr>
<td><strong>SPLIT</strong></td>
<td>split a string into two at the occurrence of a separator</td>
</tr>
</tbody>
</table>

2.7.1 The INSERT procedure: The generic function reference of the form

```
INSERT(string, start, substring)
```

shall be provided.

**Description:** Insert a substring into a string at a specified position.

**Class:** Transformational function.

**Arguments:**
- `string` - is either type `VARYING_STRING` or type default `CHARACTER`,
- `start` - is type default `INTEGER`,
- `substring` - is either type `VARYING_STRING` or type default `CHARACTER`.

All arguments must be scalars. The arguments are unchanged by the procedure.

**Result Type:** `VARYING_STRING`.

**Result Value:** The result value is a copy of the characters of the argument `string` with the characters of `substring` inserted into the copy of `string` before the character at the character position `start`. The remainder of the result string is shifted to the right and enlarged as necessary. If `start` is greater than `LEN(string)`, the value `LEN(string)+1` is used for `start` and the `substring` is appended to the copy of `string`. If `start` is less than one, the value one is used for `start` and the `substring` is inserted before the first character of the copy of the `string`. The length of the result is `LEN(string) + LEN(substring)`.

2.7.2 The REPLACE procedure: The generic function references of the forms

```
REPLACE(string, start, substring)
REPLACE(string, start, finish, substring)
REPLACE(string, target, substring, every, back)
```

shall be provided.

**Description:** Replaces a subset of the characters in a string by a given substring. The subset may be specified either by position or by content.

**Class:** Transformational function.

**Arguments:**
- `string` - is either type `VARYING_STRING` or type default `CHARACTER`,
- `start` - is type default `INTEGER`,
- `finish` - is type default `INTEGER`,
- `substring` - is either type `VARYING_STRING` or type default `CHARACTER`,
- `target` - is either type `VARYING_STRING` or type default `CHARACTER`,
- `every` - is type default `LOGICAL`, and is `OPTIONAL`,
- `back` - is type default `LOGICAL`, and is `OPTIONAL`.

All arguments are scalar and are unchanged by the procedure. The argument `target` must not be of zero length. In all cases the arguments are unchanged by the procedure.

**Result Type:** `VARYING_STRING`.
Result Value: The result value is a copy of the characters in `string` modified as per one of the cases below.

Case(i) : For a reference of the form

```plaintext
REPLACE(string, start, substring)
```

the characters of the argument `substring` are inserted in the copy of `string` beginning with the character at the character position `start`. The characters in positions from

```
s + 10 \min(s + \text{LEN}(\text{substring}) - 1, \text{LEN}(\text{string}))
```

are deleted. The result string is enlarged if necessary. If `start` is greater than `LEN(string)`, the value `LEN(string)+1` is used for `start` and `substring` is appended to the copy of `string`. If `start` is less than one, the value one is used for `start`.

Case(ii) : For a reference of the form

```plaintext
REPLACE(string, start, finish, substring)
```

the characters in the copy of `string` between positions `start` and `finish`, including those at `start` and `finish`, are deleted and replaced by the characters of `substring`. If `start` is less than one, the value one is used for `start`. If `finish` is greater than `LEN(string)`, the value `LEN(string)` is used for `finish`. If `finish` is less than `start`, the characters of `substring` are inserted before the character at `start` and no characters are deleted. The length of the result string is adjusted as necessary.

Case(iii) : For a reference of the form

```plaintext
REPLACE(string, target, substring, every, back)
```

the copy of `string` is searched for occurrences of `target`. The search is done in the backward direction if the argument `back` is present with the value true, but in the forward direction otherwise. If `target` is found, it is replaced by `substring`. If `every` is present with the value true, the search and replace is continued from the character following `target` in the search direction specified until all occurrences of `target` in the copy string are replaced; otherwise only the first occurrence of `target` is replaced.

2.7.3 The REMOVE procedure: The generic function reference of the form

```plaintext
REMOVE(string, start, finish)
```

shall be provided.

Description: Removes a specified substring from a string.

Class: Transformational function.

Arguments:

- `string` - is either type `VARYING_STRING` or type default `CHARACTER`.
- `start` - is type default `INTEGER`, and is `OPTIONAL`.
- `finish` - is type default `INTEGER`, and is `OPTIONAL`.

All arguments must be scalars. The arguments are unchanged by the procedure.

Result Type: `VARYING_STRING`.

Result Value: The result value is a copy of the characters of `string` with the characters between `start` and `finish`, inclusive, removed. If `start` is absent or less than one, the value one is used for `start`. If `finish` is absent or greater than `LEN(string)`, the value `LEN(string)` is used for `finish`. If `finish` is less than `start`, the characters of `string` are delivered unchanged as the result.
2.7.4 The EXTRACT procedure: The generic function reference of the form

```
EXTRACT(string,start,finish)
```

shall be provided.

Description: Extracts a specified substring from a string.

Class: Transformational function.

Arguments:

- `string` - is either type `VARYING_STRING` or type default `CHARACTER`,
- `start` - is type default `INTEGER`, and is `OPTIONAL`,
- `finish` - is type default `INTEGER`, and is `OPTIONAL`.

All arguments must be scalars. The arguments are unchanged by the procedure.

Result Type: `VARYING_STRING`.

Result Value: The result value is a copy of the characters of the argument `string` between `start` and `finish`, inclusive. If `start` is absent or less than one, the value one is used for `start`. If `finish` is absent or greater than `LEN(string)`, the value `LEN(string)` is used for `finish`. If `finish` is less than `start`, the result is a zero-length string.

2.7.5 The SPLIT procedure: The generic subroutine reference of the form

```
CALL SPLIT(string,word,set,separator,back)
```

shall be provided.

Description: Splits a string into a two substrings with the substrings separated by the occurrence of a character from a specified separator set.

Class: Subroutine.

Arguments:

- `string` - is type `VARYING_STRING`,
- `word` - is type `VARYING_STRING`,
- `set` - is either type `VARYING_STRING` or type default `CHARACTER`,
- `separator` - is type `VARYING_STRING`, and is `OPTIONAL`,
- `back` - is type default `LOGICAL`, and is `OPTIONAL`.

All arguments are scalar.

Action: The effect of the procedure is to divide the string at the first occurence of a character that is a member of those included in `set`. The `string` is searched in the forward direction unless `back` is present with the value true, in which case the search is in the backward direction. The characters passed over in the search are returned in the argument `word` and the remainder of the string, not including the separator character is returned in the argument `string`. If no character from `set` is found, the whole string is returned in `word` and `string` is returned as zero-length. If the argument `separator` is present, the actual character found which separates the `word` from the remainder of the `string` is returned in `separator`. The effect of the procedure is such that, on return, either

```
word//separator//string
```

is the same as the initial string for a forward search, or

```
string//separator//word
```

is the same as the initial string for a backward search.
Annex A

(Informative)

The following module is written in Fortran 90, conformant with the language as specified in the standard ISO/IEC 1539 : 1991. It is intended to be a portable implementation of a module conformant with this International Standard. It is not intended to be prescriptive of how facilities consistent with this International Standard should be provided. This module is intended primarily to demonstrate that portable facilities consistent with the interfaces and semantics required by this International Standard could be provided within the confines of the Fortran language. It is also included as a guide for users of processors which do not have supplier-provided facilities implementing this International Standard.

It should be noted that while every care has been taken by the technical working group to ensure that this module is a correct implementation of this International Standard in valid Fortran code, no guarantee is given or implied that this code will produce correct results, or even that it will execute on any particular processor. Neither is there any implication that this illustrative module is in any way an optimal implementation of this standard; it is merely one fairly straightforward portable module that is known to provide a functionally conformant implementation on a few processors.

MODULE ISO_VARYING_STRING
!
! Written by J.L.Schonfelder
! Incorporating suggestions by C.Tanasescu, C.Weber, J.Wagener and W.Walter,
! and corrections due to L.Moss, M.Cohen, P.Griffiths, B.T.Smith
! and many other members of the committee ISO/IEC JTC1/SC22/WG5
!
! Version produced (20-Jul-93)
!
!*-----------------------------------------------------------------------------*
!* This module defines the interface and one possible implementation for a *
!* dynamic length character string facility in Fortran 90. The Fortran 90 *
!* language is defined by the standard ISO/IEC 1539 : 1991. *
!* The publicly accessible interface defined by this module is conformant *
!* with the auxiliary standard, ISO/IEC 1539-1 : 1993. *
!* The detailed implementation may be considered as an informal definition of *
!* the required semantics, and may also be used as a guide to the production *
!* of a portable implementation. *
!* N.B. Although every care has been taken to produce valid Fortran code in *
!* construction of this module no guarantee is given or implied that this *
!* code will work correctly without error on any specific processor, nor *
!* is this implementation intended to be in any way optimal either in use *
!* of storage or CPU cycles. *
!*-----------------------------------------------------------------------------*
!
! PRIVATE
!*-----------------------------------------------------------------------------*
!* By default all entities declared or defined in this module are private to *
!* the module. Only those entities declared explicitly as being public are *
!* accessible to programs using the module. In particular, the procedures and *
!* operators defined herein are made accessible via their generic identifiers *
!* only; their specific names are private. *
!*-----------------------------------------------------------------------------*
!
! TYPE VARYING_STRING
! PRIVATE
! CHARACTER,DIMENSION(:),POINTER :: chars
ENDTYPE VARYING_STRING
!
!*-----------------------------------------------------------------------------*
!* The representation chosen for this definition of the module is of a string *
!* type consisting of a single component that is a pointer to a rank one array *
!* of characters. *
! Note: this Module is defined only for characters of default kind. A similar !
! module could be defined for non-default characters if these are supported !
! on a processor by adding a KIND parameter to the component in the type !
! definition, and to all declarations of objects of CHARACTER type. !

CHARACTER, PARAMETER :: blank = " "

!----- GENERIC PROCEDURE INTERFACE DEFINITIONS ----------------------------------!

!----- LEN interface ----------------------------------------------------------!
INTERFACE LEN
  MODULE PROCEDURE len_s  ! length of string
ENDINTERFACE

!----- Conversion procedure interfaces ---------------------------------------!
INTERFACE VAR_STR
  MODULE PROCEDURE c_to_s  ! character to string
  MODULE PROCEDURE s_to_c, & ! string to character
  MODULE PROCEDURE s_to_fix_c ! string to specified length character
ENDINTERFACE

!----- ASSIGNMENT interfaces --------------------------------------------------!
INTERFACE ASSIGNMENT(=)
  MODULE PROCEDURE s_ass_s, & ! string = string
  MODULE PROCEDURE c_ass_s, & ! character = string
  MODULE PROCEDURE s_ass_c ! string = character
ENDINTERFACE

!----- Concatenation operator interfaces -------------------------------------!
INTERFACE OPERATOR(/)
  MODULE PROCEDURE s_concat_s, & ! string//string
  MODULE PROCEDURE s_concat_c, & ! string//character
  MODULE PROCEDURE c_concat_s ! character//string
ENDINTERFACE

!----- Repeated Concatenation interfaces --------------------------------------!
INTERFACE REPEAT
  MODULE PROCEDURE repeat_s
ENDINTERFACE

!----- Equality comparison operator interfaces -------------------------------!
INTERFACE OPERATOR(==)
  MODULE PROCEDURE s_eq_s, & ! string==string
  MODULE PROCEDURE s_eq_c, & ! string==character
  MODULE PROCEDURE c_eq_s ! character==string
ENDINTERFACE

!----- not-equality comparison operator interfaces ----------------------------!
INTERFACE OPERATOR(/=)
  MODULE PROCEDURE s_ne_s, & ! string/=string
  MODULE PROCEDURE s_ne_c, & ! string/=character
  MODULE PROCEDURE c_ne_s ! character/=string
ENDINTERFACE

!----- less-than comparison operator interfaces ------------------------------!
INTERFACE OPERATOR(<)
  MODULE PROCEDURE s_lt_s, & ! string<string
  MODULE PROCEDURE s_lt_c, & ! string<character
  MODULE PROCEDURE c_lt_s ! character<string
ENDINTERFACE

!----- less-than-or-equal comparison operator interfaces -----------------------!
INTERFACE OPERATOR(<=)
  MODULE PROCEDURE s_le_s, & ! string<=string
  MODULE PROCEDURE s_le_c, & ! string<=character
  MODULE PROCEDURE c_le_s ! character<=string
ENDINTERFACE
MODULE PROCEDURE s_ge_s, & ! string>=string
  s_ge_c, & ! string>=character
  c_ge_s ! character>=string
ENDINTERFACE

INTERFACE OPERATOR(>)
  MODULE PROCEDURE s_gt_s, & ! string>string
  s_gt_c, & ! string>character
  c_gt_s ! character>string
ENDINTERFACE

INTERFACE LLT
  MODULE PROCEDURE s_llt_s, & ! LLT(string,string)
  s_llt_c, & ! LLT(string,character)
  c_llt_s ! LLT(character,string)
ENDINTERFACE

INTERFACE LLE
  MODULE PROCEDURE s_lle_s, & ! LLE(string,string)
  s_lle_c, & ! LLE(string,character)
  c_lle_s ! LLE(character,string)
ENDINTERFACE

INTERFACE LGE
  MODULE PROCEDURE s_lge_s, & ! LGE(string,string)
  s_lge_c, & ! LGE(string,character)
  c_lge_s ! LGE(character,string)
ENDINTERFACE

INTERFACE LGT
  MODULE PROCEDURE s_lgt_s, & ! LGT(string,string)
  s_lgt_c, & ! LGT(string,character)
  c_lgt_s ! LGT(character,string)
ENDINTERFACE

INTERFACE GET
  MODULE PROCEDURE get_d_eor, & ! default unit, EoR termination
  get_u_eor, & ! specified unit, EoR termination
  get_d_tset_s, & ! default unit, string set termination
  get_u_tset_s, & ! specified unit, string set termination
  get_d_tset_c, & ! default unit, char set termination
  get_u_tset_c ! specified unit, char set termination
ENDINTERFACE

INTERFACE PUT
  MODULE PROCEDURE put_d_s, & ! string to default unit
  put_u_s, & ! string to specified unit
  put_d_c, & ! char to default unit
  put_u_c ! char to specified unit
ENDINTERFACE

INTERFACE PUT_LINE
  MODULE PROCEDURE putline_d_s, & ! string to default unit
  putline_u_s, & ! string to specified unit
  putline_d_c, & ! char to default unit
  putline_u_c ! char to specified unit
ENDINTERFACE

INTERFACE INSERT
  MODULE PROCEDURE insert_ss, & ! string in string
  insert_sc, & ! char in string
  insert_cs, & ! string in char
ENDINTERFACE

!----- Replace procedure interfaces ------------------------------------------!
INTERFACE REPLACE
    MODULE PROCEDURE replace_ss, & ! string by string, at specified
        replace_sc, & ! string by char , starting
        replace_cc, & ! char by string , point
        replace_cc_s, & ! char by char
        replace_ss_sf, & ! string by string, between
        replace_sc_sf, & ! string by char , specified
        replace_sss, & ! in string replace string by string
        replace_ssc, & ! in string replace string by char
        replace_ssc_s, & ! in string replace char by string
        replace_scc, & ! in string replace char by char
        replace_scc_s, & ! in string replace char by char, starting and
        replace_cc, & ! char by char , finishing points
        replace_sss_s, & ! in string replace string by string
        replace_ssc_s, & ! in string replace string by char
        replace_scc_s, & ! in char replace char by string
        replace_scc_s, & ! in char replace char by char
    ENDINTERFACE

ENDINTERFACE

!----- Remove procedure interface ------------------------------------------!
INTERFACE REMOVE
    MODULE PROCEDURE remove_s, & ! characters from string, between start
        remove_c ! characters from char , and finish
    ENDINTERFACE

ENDINTERFACE

!----- Extract procedure interface ------------------------------------------!
INTERFACE EXTRACT
    MODULE PROCEDURE extract_s, & ! from string extract string, between start
        extract_c ! from char extract string, and finish
    ENDINTERFACE

ENDINTERFACE

!----- Split procedure interface ------------------------------------------!
INTERFACE SPLIT
    MODULE PROCEDURE split_s, & ! split string at first occurrence of
        split_c ! character in set
    ENDINTERFACE

ENDINTERFACE

!----- Index procedure interfaces ------------------------------------------!
INTERFACE INDEX
    MODULE PROCEDURE index_ss, index_sc, index_cs
    ENDINTERFACE

ENDINTERFACE

!----- Scan procedure interfaces ------------------------------------------!
INTERFACE SCAN
    MODULE PROCEDURE scan_ss, scan_sc, scan_cs
    ENDINTERFACE

ENDINTERFACE

!----- Verify procedure interfaces ------------------------------------------!
INTERFACE VERIFY
    MODULE PROCEDURE verify_ss, verify_sc, verify_cs
    ENDINTERFACE

ENDINTERFACE

!----- Interfaces for remaining intrinsic function overloads -------------------!
INTERFACE LEN_TRIM
    MODULE PROCEDURE len_trim_s
    ENDINTERFACE

ENDINTERFACE

INTERFACE TRIM
    MODULE PROCEDURE trim_s
    ENDINTERFACE

INTERFACE IACHAR
    MODULE PROCEDURE iachar_s
    ENDINTERFACE

INTERFACE ICHAR
    MODULE PROCEDURE ichar_s
    ENDINTERFACE

ENDINTERFACE
INTERFACE ADJUSTL
  MODULE PROCEDURE adjustl_s
ENDINTERFACE

INTERFACE ADJUSTR
  MODULE PROCEDURE adjustr_s
ENDINTERFACE

!----- specification of publically accessible entities -----------------------!
PUBLIC :: VARYING_STRING, VAR_STR, CHAR, LEN, GET, PUT, PUT_LINE, INSERT, REPLACE, &
SPLIT, REMOVE, REPEAT, EXTRACT, INDEX, SCAN, VERIFY, LLT, LLE, LGE, LGT, &
ASSIGNMENT (=), OPERATOR (/\), OPERATOR (==), OPERATOR (/\=), OPERATOR (<), &
OPERATOR (<=), OPERATOR (>=), OPERATOR (>), LEN_TRIM, TRIM, IACHAR, ICHAR, &
ADJUSTL, ADJUSTR

CONTAINS

!----- LEN Procedure ---------------------------------------------------------!
FUNCTION len_s(string)
type(VARYING_STRING), INTENT(IN) :: string
INTEGER :: len_s
! returns the length of the string argument or zero if there is no current
! string value
IF (.NOT. ASSOCIATED(string%chars)) THEN
  len_s = 0
ELSE
  len_s = SIZE(string%chars)
ENDIF
ENDFUNCTION len_s

!----- Conversion Procedures ------------------------------------------------!
FUNCTION c_to_s(chr)
type(VARYING_STRING) :: c_to_s
CHARACTER(LEN=*) :: chr
! returns the string consisting of the characters char
INTEGER :: lc
lc=LEN(chr)
ALLOCATE(c_to_s%chars(1:lc))
DO i=1,lc
  c_to_s%chars(i) = chr(i:i)
ENDDO
ENDFUNCTION c_to_s

FUNCTION s_to_c(string)
type(VARYING_STRING), INTENT(IN) :: string
CHARACTER(LEN=SIZE(string%chars)) :: s_to_c
! returns the characters of string as an automatically sized character
INTEGER :: lc
lc=SIZE(string%chars)
DO i=1,lc
  s_to_c(i:i) = string%chars(i)
ENDDO
ENDFUNCTION s_to_c

FUNCTION s_to_fix_c(string,length)
type(VARYING_STRING), INTENT(IN) :: string
INTEGER, INTENT(IN) :: length
CHARACTER(LEN=length) :: s_to_fix_c
! returns the character of fixed length, length, containing the characters
! of string either padded with blanks or truncated on the right to fit
INTEGER :: lc
lc=MIN(SIZE(string%chars),length)
DO i=1,lc
  s_to_fix_c(i:i) = string%chars(i)
ENDDO
ENDFUNCTION s_to_fix_c

!----- ASSIGNMENT Procedures -------------------------------------------------!

1
2 INTERFACE ADJUSTL
3   MODULE PROCEDURE adjustl_s
4 ENDINTERFACE
5
6 INTERFACE ADJUSTR
7   MODULE PROCEDURE adjustr_s
8 ENDINTERFACE
9
10 !----- specification of publically accessible entities -----------------------!
11 PUBLIC :: VARYING_STRING, VAR_STR, CHAR, LEN, GET, PUT, PUT_LINE, INSERT, REPLACE, &
12 SPLIT, REMOVE, REPEAT, EXTRACT, INDEX, SCAN, VERIFY, LLT, LLE, LGE, LGT, &
13 ASSIGNMENT (=), OPERATOR (/\), OPERATOR (==), OPERATOR (/\=), OPERATOR (<), &
14 OPERATOR (<=), OPERATOR (>=), OPERATOR (>), LEN_TRIM, TRIM, IACHAR, ICHAR, &
15 ADJUSTL, ADJUSTR
16
17 CONTAINS
18
19 !----- LEN Procedure ---------------------------------------------------------!
20 FUNCTION len_s(string)
21 type(VARYING_STRING), INTENT(IN) :: string
22 INTEGER :: len_s
23 ! returns the length of the string argument or zero if there is no current
24 ! string value
25 IF (.NOT. ASSOCIATED(string%chars)) THEN
26   len_s = 0
27 ELSE
28   len_s = SIZE(string%chars)
29 ENDIF
30 ENDFUNCTION len_s
31
32 !----- Conversion Procedures ------------------------------------------------!
33 FUNCTION c_to_s(chr)
34 type(VARYING_STRING) :: c_to_s
35 CHARACTER(LEN=*) :: chr
36 ! returns the string consisting of the characters char
37 INTEGER :: lc
38 lc=LEN(chr)
39 ALLOCATE(c_to_s%chars(1:lc))
40 DO i=1,lc
41   c_to_s%chars(i) = chr(i:i)
42 ENDDO
43 ENDFUNCTION c_to_s
44
45 FUNCTION s_to_c(string)
46 type(VARYING_STRING), INTENT(IN) :: string
47 CHARACTER(LEN=SIZE(string%chars)) :: s_to_c
48 ! returns the characters of string as an automatically sized character
49 INTEGER :: lc
50 lc=SIZE(string%chars)
51 DO i=1,lc
52   s_to_c(i:i) = string%chars(i)
53 ENDDO
54 ENDFUNCTION s_to_c
55
56 FUNCTION s_to_fix_c(string,length)
57 type(VARYING_STRING), INTENT(IN) :: string
58 INTEGER, INTENT(IN) :: length
59 CHARACTER(LEN=length) :: s_to_fix_c
60 ! returns the character of fixed length, length, containing the characters
61 ! of string either padded with blanks or truncated on the right to fit
62 INTEGER :: lc
63 lc=MIN(SIZE(string%chars),length)
64 DO i=1,lc
65   s_to_fix_c(i:i) = string%chars(i)
66 ENDDO
67 IF (lc < length) THEN  ! result longer than string padding needed
68   s_to_fix_c(lc+1:length) = blank
69 ENDIF
70 ENDFUNCTION s_to_fix_c
71
72 !----- ASSIGNMENT Procedures -------------------------------------------------!
73
19
SUBROUTINE s_ass_s(var,expr)
  type(VARYING_STRING), INTENT(OUT) :: var
  type(VARYING_STRING), INTENT(IN) :: expr
  ! assign a string value to a string variable overriding default assignment
  ! reallocates string variable to size of string value and copies characters
  ALLOCATE(var%chars(1:LEN(expr)))
  var%chars = expr%chars
ENDSUBROUTINE s_ass_s

SUBROUTINE c_ass_s(var,expr)
  CHARACTER(LEN=*), INTENT(OUT) :: var
  type(VARYING_STRING), INTENT(IN) :: expr
  ! assign a string value to a character variable
  ! if the string is longer than the character truncate the string on the right
  ! if the string is shorter the character is blank padded on the right
  INTEGER :: lc, ls
  lc = LEN(var); ls = MIN(LEN(expr), lc)
  DO i = 1, ls
    var(i:i) = expr%chars(i)
  ENDDO
  DO i = ls+1, lc
    var(i:i) = blank
  ENDDO
ENDSUBROUTINE c_ass_s

SUBROUTINE s_ass_c(var,expr)
  type(VARYING_STRING), INTENT(OUT) :: var
  CHARACTER(LEN=*), INTENT(IN) :: expr
  ! assign a character value to a string variable
  ! disassociates the string variable from its current value, allocates new
  ! space to hold the characters and copies them from the character value
  ! into this space.
  INTEGER :: lc
  lc = LEN(expr)
  ALLOCATE(var%chars(1:lc))
  DO i = 1, lc
    var%chars(i) = expr(i:i)
  ENDDO
ENDSUBROUTINE s_ass_c

!----- Concatenation operator procedures ------------------------------------!
FUNCTION s_concat_s(string_a,string_b) ! string//string
  type(VARYING_STRING), INTENT(IN) :: string_a,string_b
  type(VARYING_STRING) :: s_concat_s
  INTEGER :: la, lb
  la = LEN(string_a); lb = LEN(string_b)
  ALLOCATE(s_concat_s%chars(1:la+lb))
  s_concat_s%chars(1:la) = string_a%chars
  s_concat_s%chars(1+la:la+lb) = string_b%chars
ENDFUNCTION s_concat_s

FUNCTION s_concat_c(string_a,string_b) ! string//character
  type(VARYING_STRING), INTENT(IN) :: string_a
  CHARACTER(LEN=*), INTENT(IN) :: string_b
  type(VARYING_STRING) :: s_concat_c
  INTEGER :: la, lb
  la = LEN(string_a); lb = LEN(string_b)
  ALLOCATE(s_concat_c%chars(1:la+lb))
  s_concat_c%chars(1:la) = string_a%chars
  s_concat_c%chars(1+la:1+la+lb) = string_b%chars
ENDFUNCTION s_concat_c

FUNCTION c_concat_s(string_a,string_b) ! character//string
  CHARACTER(LEN=*), INTENT(IN) :: string_a
  type(VARYING_STRING), INTENT(IN) :: string_b
  type(VARYING_STRING) :: c_concat_s
  INTEGER :: la, lb
  la = LEN(string_a); lb = LEN(string_b)
  ALLOCATE(c_concat_s%chars(1:la+lb))
  DO i = 1, la
    c_concat_s%chars(la+i) = string_b(i:i)
  ENDDO
ENDFUNCTION c_concat_s

FUNCTION c_concat_c(string_a,string_b) ! character//character
  CHARACTER(LEN=*), INTENT(IN) :: string_a
  type(VARYING_STRING), INTENT(IN) :: string_b
  type(VARYING_STRING) :: c_concat_c
  INTEGER :: la, lb
  la = LEN(string_a); lb = LEN(string_b)
  ALLOCATE(c_concat_c%chars(1:la+lb))
  DO i = 1, lb
    c_concat_c%chars(la+i) = string_b(i:i)
  ENDDO
ENDFUNCTION c_concat_c

!----- Concatenation operator procedures ------------------------------------!
c_concat_s%chars(i) = string_a(i:i)
ENDDO

!---- Repeated concatenation procedures ---------------------------------------!
FUNCTION repeat_s(string,ncopies)
type(VARYING_STRING),INTENT(IN) :: string
INTEGER,INTENT(IN) :: ncopies
type(VARYING_STRING) :: repeat_s
!
INTERFACE
  ! Returns a string produced by the concatenation of ncopies of the
  ! argument string
  INTEGER :: lr,ls
  IF (ncopies < 0) THEN
    WRITE(*,*) " Negative ncopies requested in REPEAT"$
    STOP
  ENDIF
ls = LEN(string); lr = ls*ncopies
ALLOCATE(repeat_s%chars(1:lr))
DO i = 1,ncopies
  repeat_s%chars(1+(i-1)*ls:i*ls) = string%chars
ENDDO
ENDFUNCTION repeat_s

!------ Equality comparison operators ----------------------------------------!
FUNCTION s_eq_s(string_a,string_b) ! string==string
  type(VARYING_STRING),INTENT(IN) :: string_a,string_b
  LOGICAL :: s_eq_s
  INTEGER :: la,lb
  la = LEN(string_a); lb = LEN(string_b)
  IF (la > lb) THEN
    s_eq_s = ALL(string_a%chars(1:lb) == string_b%chars) .AND. 
             ALL(string_a%chars(lb+1:la) == blank)
  ELSEIF (la < lb) THEN
    s_eq_s = ALL(string_a%chars == string_b%chars(1:la)) .AND. 
             ALL(blank == string_b%chars(la+1:lb))
  ELSE
    s_eq_s = ALL(string_a%chars == string_b%chars)
  ENDIF
ENDFUNCTION s_eq_s

FUNCTION s_eq_c(string_a,string_b) ! string==character
  CHARACTER(LEN=*) ,INTENT(IN) :: string_a
  type(VARYING_STRING),INTENT(IN) :: string_b
  LOGICAL :: s_eq_c
  INTEGER :: la,lb,ls
  la = LEN(string_a); lb = LEN(string_b); ls = MIN(la,lb)
  DO i = 1,ls
    IF( string_a%chars(i) /= string_b(i:i) )THEN
      s_eq_c = .FALSE.; RETURN
    ENDIF
  ENDDO
  IF( la > lb .AND. ANY( string_a%chars(lb+1:la) /= blank ) )THEN
    s_eq_c = .FALSE.; RETURN
  ELSEIF( la < lb .AND. blank /= string_b(la+1:lb) )THEN
    s_eq_c = .FALSE.; RETURN
  ENDIF
  s_eq_c = .TRUE.
ENDFUNCTION s_eq_c

FUNCTION c_eq_s(string_a,string_b) ! character==string
  CHARACTER(LEN=*) ,INTENT(IN) :: string_a
  type(VARYING_STRING),INTENT(IN) :: string_b
  LOGICAL :: c_eq_s
  INTEGER :: la,lb,ls
  la = LEN(string_a); lb = LEN(string_b); ls = MIN(la,lb)
  DO i = 1,ls
    IF( string_a(i:i) /= string_b%chars(i) )THEN
      c_eq_s = .FALSE.; RETURN
    ENDIF
  ENDDO
  IF( la > lb .AND. string_a(lb+1:la) /= blank )THEN
  ELSE
c_eq_s = .FALSE.; RETURN
ELSEIF( la < lb .AND. ANY( blank /= string_b%chars(la+1:lb) ) ) THEN
  c_eq_s = .FALSE.; RETURN
ENDIF

FUNCTION c_eq_s

!------ Non-equality operators -----------------------------------------------!

FUNCTION s_ne_s(string_a,string_b) ! string/=string
  type(VARYING_STRING),INTENT(IN) :: string_a,string_b
  LOGICAL :: s_ne_s
  INTEGER :: la,lb
  la = LEN(string_a); lb = LEN(string_b)
  IF (la > lb) THEN
    s_ne_s = ANY(string_a%chars(1:lb) /= string_b%chars) .OR. &
          ANY(string_a%chars(lb+1:la) /= blank)
  ELSEIF (la < lb) THEN
    s_ne_s = ANY(string_a%chars /= string_b%chars(1:la)) .OR. &
          ANY(blank /= string_b%chars(la+1:lb))
  ELSE
    s_ne_s = ANY(string_a%chars /= string_b%chars)
  ENDIF
ENDFUNCTION s_ne_s

FUNCTION s_ne_c(string_a,string_b) ! string/=character
  type(VARYING_STRING),INTENT(IN) :: string_a
  CHARACTER(LEN=*) ,INTENT(IN) :: string_b
  LOGICAL :: s_ne_c
  INTEGER :: la,lb,ls
  la = LEN(string_a); lb = LEN(string_b); ls = MIN(la,lb)
  DO i = 1,ls
    IF( string_a%chars(i) /= string_b(i:i) ) THEN
      s_ne_c = .TRUE.; RETURN
    ENDIF
  ENDDO
  IF( la > lb .AND. ANY( string_a%chars(lb+1:la) /= blank ) ) THEN
    s_ne_c = .TRUE.; RETURN
  ELSEIF( la < lb .AND. blank /= string_b(la+1:lb) ) THEN
    s_ne_c = .TRUE.; RETURN
  ENDIF
  s_ne_c = .FALSE.
ENDFUNCTION s_ne_c

FUNCTION c_ne_s(string_a,string_b) ! character/=string
  CHARACTER(LEN=*) ,INTENT(IN) :: string_a
  type(VARYING_STRING),INTENT(IN) :: string_b
  LOGICAL :: c_ne_s
  INTEGER :: la,lb,ls
  la = LEN(string_a); lb = LEN(string_b); ls = MIN(la,lb)
  DO i = 1,ls
    IF( string_a(i:i) /= string_b%chars(i) ) THEN
      c_ne_s = .TRUE.; RETURN
    ENDIF
  ENDDO
  IF( la > lb .AND. string_a(lb+1:la) /= blank ) THEN
    c_ne_s = .TRUE.; RETURN
  ELSEIF( la < lb .AND. ANY( blank /= string_b%chars(la+1:lb) ) ) THEN
    c_ne_s = .TRUE.; RETURN
  ENDIF
  c_ne_s = .FALSE.
ENDFUNCTION c_ne_s

!------ Less-than operators --------------------------------------------------!

FUNCTION s_lt_s(string_a,string_b) ! string<string
  type(VARYING_STRING),INTENT(IN) :: string_a,string_b
  LOGICAL :: s_lt_s
  INTEGER :: ls,la,lb
  la = LEN(string_a); lb = LEN(string_b); ls = MIN(la,lb)
  DO i = 1,ls
    IF( string_a%chars(i) < string_b%chars(i) ) THEN
      s_lt_s = .TRUE.; RETURN
    ENDIF
  ENDDO
  IF( la > lb .AND. string_a(lb+1:la) /= blank ) THEN
    s_lt_s = .TRUE.; RETURN
  ELSEIF( la < lb .AND. ANY( blank /= string_b%chars(la+1:lb) ) ) THEN
    s_lt_s = .TRUE.; RETURN
  ENDIF
  s_lt_s = .FALSE.
ENDFUNCTION s_lt_s

FUNCTION c_lt_s(string_a,string_b) ! character<string
  CHARACTER(LEN=*) ,INTENT(IN) :: string_a
  type(VARYING_STRING),INTENT(IN) :: string_b
  LOGICAL :: c_lt_s
  INTEGER :: la,lb,ls
  la = LEN(string_a); lb = LEN(string_b); ls = MIN(la,lb)
  DO i = 1,ls
    IF( string_a(i:i) < string_b%chars(i) ) THEN
      c_lt_s = .TRUE.; RETURN
    ENDIF
  ENDDO
  IF( la > lb .AND. string_a(lb+1:la) /= blank ) THEN
    c_lt_s = .TRUE.; RETURN
  ELSEIF( la < lb .AND. ANY( blank /= string_b%chars(la+1:lb) ) ) THEN
    c_lt_s = .TRUE.; RETURN
  ENDIF
  c_lt_s = .FALSE.
ENDFUNCTION c_lt_s
FUNCTION s_lt_s(string_a,string_b) ! string<character
  type(VARYING_STRING),INTENT(IN) :: string_a
  CHARACTER(LEN=*),INTENT(IN) :: string_b
  LOGICAL :: s_lt_s
  INTEGER :: ls,la,lb
  la = LEN(string_a); lb = LEN(string_b); ls = MIN(la,lb)
  DO i = 1,ls
    IF( string_a%chars(i) < string_b%chars(i) )THEN
      s_lt_s = .TRUE.; RETURN
    ELSEIF( string_a%chars(i) > string_b%chars(i) )THEN
      s_lt_s = .FALSE.; RETURN
    ENDIF
  ENDDO
  IF( la < lb )THEN
    DO i = la+1,lb
      IF( blank < string_b%chars(i) )THEN
        s_lt_s = .TRUE.; RETURN
      ELSEIF( blank > string_b%chars(i) )THEN
        s_lt_s = .FALSE.; RETURN
      ENDIF
    ENDDO
  ELSEIF( la > lb )THEN
    DO i = lb+1,la
      IF( string_a%chars(i) < blank )THEN
        s_lt_s = .TRUE.; RETURN
      ELSEIF( string_a%chars(i) > blank )THEN
        s_lt_s = .FALSE.; RETURN
      ENDIF
    ENDDO
  ENDIF
  s_lt_s = .FALSE.
ENDFUNCTION s_lt_s

FUNCTION s_lt_c(string_a,string_b) ! string<character
  type(VARYING_STRING),INTENT(IN) :: string_a
  CHARACTER(LEN=*),INTENT(IN) :: string_b
  LOGICAL :: s_lt_c
  INTEGER :: ls,la,lb
  la = LEN(string_a); lb = LEN(string_b); ls = MIN(la,lb)
  DO i = 1,ls
    IF( string_a%chars(i) < string_b%chars(i) )THEN
      s_lt_c = .TRUE.; RETURN
    ELSEIF( string_a%chars(i) > string_b%chars(i) )THEN
      s_lt_c = .FALSE.; RETURN
    ENDIF
  ENDDO
  IF( la < lb )THEN
    DO i = la+1,lb
      IF( blank < string_b%chars(i) )THEN
        s_lt_c = .TRUE.; RETURN
      ELSEIF( blank > string_b%chars(i) )THEN
        s_lt_c = .FALSE.; RETURN
      ENDIF
    ENDDO
  ELSEIF( la > lb )THEN
    DO i = lb+1,la
      IF( string_a%chars(i) < blank )THEN
        s_lt_c = .TRUE.; RETURN
      ELSEIF( string_a%chars(i) > blank )THEN
        s_lt_c = .FALSE.; RETURN
      ENDIF
    ENDDO
  ENDIF
  s_lt_c = .FALSE.
ENDFUNCTION s_lt_c

FUNCTION c_lt_s(string_a,string_b) ! character<string
  CHARACTER(LEN=*),INTENT(IN) :: string_a
  type(VARYING_STRING),INTENT(IN) :: string_b
  LOGICAL :: c_lt_s
  INTEGER :: ls,la,lb
  la = LEN(string_a); lb = LEN(string_b); ls = MIN(la,lb)
  DO i = 1,ls
    IF( string_a%chars(i:i) < string_b%chars(i:i) )THEN
      c_lt_s = .TRUE.; RETURN
    ELSEIF( string_a%chars(i:i) > string_b%chars(i:i) )THEN
      c_lt_s = .FALSE.; RETURN
    ENDIF
  ENDDO
  IF( la < lb )THEN
    DO i = la+1,lb
      IF( blank < string_b%chars(i) )THEN
        c_lt_s = .TRUE.; RETURN
      ELSEIF( blank > string_b%chars(i) )THEN
        c_lt_s = .FALSE.; RETURN
      ENDIF
    ENDDO
  ELSEIF( la > lb )THEN
    DO i = lb+1,la
      IF( string_a%chars(i) < blank )THEN
        c_lt_s = .TRUE.; RETURN
      ELSEIF( string_a%chars(i) > blank )THEN
        c_lt_s = .FALSE.; RETURN
      ENDIF
    ENDDO
  ENDIF
  c_lt_s = .FALSE.
ENDFUNCTION c_lt_s
c_lt_s = .FALSE.; RETURN
ENDIF
ELSEIF( la > lb )THEN
IF( string_a(lb+1:la) < blank )THEN
c_lt_s = .TRUE.; RETURN
ELSEIF( string_a(lb+1:la) > blank )THEN
c_lt_s = .FALSE.; RETURN
ENDIF
ENDIF
c_lt_s = .FALSE.
ENDFUNCTION c_lt_s

!------ Less-than-or-equal-to operators --------------------------------------!
FUNCTION s_le_s(string_a,string_b) ! string<=string
  type(VARYING_STRING),INTENT(IN) :: string_a,string_b
  LOGICAL :: s_le_s
  INTEGER :: ls,la,lb
  la = LEN(string_a); lb = LEN(string_b); ls = MIN(la,lb)
  DO i = 1,ls
    IF( string_a%chars(i) < string_b%chars(i) )THEN
      s_le_s = .TRUE.; RETURN
    ELSEIF( string_a%chars(i) > string_b%chars(i) )THEN
      s_le_s = .FALSE.; RETURN
    ENDIF
  ENDDO
  IF( la < lb )THEN
    DO i = la+1,lb
      IF( blank < string_b%chars(i) )THEN
        s_le_s = .TRUE.; RETURN
      ELSEIF( blank > string_b%chars(i) )THEN
        s_le_s = .FALSE.; RETURN
      ENDIF
    ENDDO
  ELSEIF( la > lb )THEN
    DO i = lb+1,la
      IF( string_a%chars(i) < blank )THEN
        s_le_s = .TRUE.; RETURN
      ELSEIF( string_a%chars(i) > blank )THEN
        s_le_s = .FALSE.; RETURN
      ENDIF
    ENDDO
  ENDIF
  s_le_s = .TRUE.
ENDFUNCTION s_le_s

FUNCTION s_le_c(string_a,string_b) ! string<=character
  type(VARYING_STRING),INTENT(IN) :: string_a
  CHARACTER(LEN=*),INTENT(IN) :: string_b
  LOGICAL :: s_le_c
  INTEGER :: ls,la,lb
  la = LEN(string_a); lb = LEN(string_b); ls = MIN(la,lb)
  DO i = 1,ls
    IF( string_a%chars(i) < string_b(i:i) )THEN
      s_le_c = .TRUE.; RETURN
    ELSEIF( string_a%chars(i) > string_b(i:i) )THEN
      s_le_c = .FALSE.; RETURN
    ENDIF
  ENDDO
  IF( la < lb )THEN
    IF( blank < string_b(la+1:lb) )THEN
      s_le_c = .TRUE.; RETURN
    ELSEIF( blank > string_b(la+1:lb) )THEN
      s_le_c = .FALSE.; RETURN
    ENDIF
  ELSEIF( la > lb )THEN
    DO i = lb+1,la
      IF( string_a%chars(i) < blank )THEN
        s_le_c = .TRUE.; RETURN
      ELSEIF( string_a%chars(i) > blank )THEN
        s_le_c = .FALSE.; RETURN
      ENDIF
    ENDDO
  ENDIF
  s_le_c = .TRUE.
ENDFUNCTION s_le_c
FUNCTION s_le_c(string_a,string_b) ! character<=string
  CHARACTER(LEN=*) , INTENT(IN) :: string_a
  type(VARYING_STRING) , INTENT(IN) :: string_b
  LOGICAL :: c_le_s
  INTEGER :: ls,la,lb
  la = LEN(string_a); lb = LEN(string_b); ls = MIN(la,lb)
  DO i = 1,ls
    IF( string_a(i:i) < string_b%chars(i) )THEN
      c_le_s = .TRUE.; RETURN
    ELSEIF( string_a(i:i) > string_b%chars(i) )THEN
      c_le_s = .FALSE.; RETURN
    ENDIF
  ENDDO
  IF( la < lb )THEN
    DO i = la+1,lb
      IF( blank < string_b%chars(i) )THEN
        c_le_s = .TRUE.; RETURN
      ELSEIF( blank > string_b%chars(i) )THEN
        c_le_s = .FALSE.; RETURN
      ENDIF
    ENDDO
  ELSEIF( la > lb )THEN
    IF( string_a(lb+1:la) < blank )THEN
      c_le_s = .TRUE.; RETURN
    ELSEIF( string_a(lb+1:la) > blank )THEN
      c_le_s = .FALSE.; RETURN
    ENDIF
  ENDIF
c_le_s = .TRUE.
ENDFUNCTION s_le_c

FUNCTION c_le_s(string_a,string_b) ! character<=string
  CHARACTER(LEN=*),INTENT(IN) :: string_a
  type(VARYING_STRING),INTENT(IN) :: string_b
  LOGICAL :: c_le_s
  INTEGER :: ls,la,lb
  la = LEN(string_a); lb = LEN(string_b); ls = MIN(la,lb)
  DO i = 1,ls
    IF( string_a(i:i) < string_b%chars(i) )THEN
      c_le_s = .TRUE.; RETURN
    ELSEIF( string_a(i:i) > string_b%chars(i) )THEN
      c_le_s = .FALSE.; RETURN
   ENDIF
  ENDDO
  IF( la < lb )THEN
    DO i = la+1,lb
      IF( blank < string_b%chars(i) )THEN
        c_le_s = .TRUE.; RETURN
      ELSEIF( blank > string_b%chars(i) )THEN
        c_le_s = .FALSE.; RETURN
      ENDIF
    ENDDO
  ELSEIF( la > lb )THEN
    IF( string_a(lb+1:la) < blank )THEN
      c_le_s = .TRUE.; RETURN
    ELSEIF( string_a(lb+1:la) > blank )THEN
      c_le_s = .FALSE.; RETURN
    ENDIF
  ENDIF
  c_le_s = .TRUE.
ENDFUNCTION c_le_s

FUNCTION s_ge_s(string_a,string_b) ! string>=string
  type(VARYING_STRING),INTENT(IN) :: string_a,string_b
  LOGICAL :: s_ge_s
  INTEGER :: ls,la,lb
  la = LEN(string_a); lb = LEN(string_b); ls = MIN(la,lb)
  DO i = 1,ls
    IF( string_a%chars(i) > string_b%chars(i) )THEN
      s_ge_s = .TRUE.; RETURN
    ELSEIF( string_a%chars(i) < string_b%chars(i) )THEN
      s_ge_s = .FALSE.; RETURN
   ENDIF
  ENDDO
  IF( la < lb )THEN
    DO i = la+1,lb
      IF( blank > string_b%chars(i) )THEN
        s_ge_s = .TRUE.; RETURN
      ELSEIF( blank < string_b%chars(i) )THEN
        s_ge_s = .FALSE.; RETURN
     ENDIF
    ENDDO
  ELSEIF( la > lb )THEN
    DO i = lb+1,la
      IF( string_a%chars(i) > blank )THEN
        s_ge_s = .TRUE.; RETURN
      ELSEIF( string_a%chars(i) < blank )THEN
        s_ge_s = .FALSE.; RETURN
      ENDIF
    ENDDO
  ENDIF
s_ge_s = .TRUE.
ENDFUNCTION s_ge_s

FUNCTION s_ge_c(string_a,string_b) ! string>=character
  type(VARYING_STRING),INTENT(IN) :: string_a
  CHARACTER(LEN=*) , INTENT(IN) :: string_b
  LOGICAL :: s_ge_c
  INTEGER :: ls,la,lb
  la = LEN(string_a); lb = LEN(string_b); ls = MIN(la,lb)
  DO i = 1,ls
    IF( string_a(i:i) > string_b%chars(i) )THEN
      s_ge_c = .TRUE.; RETURN
    ELSEIF( string_a(i:i) < string_b%chars(i) )THEN
      s_ge_c = .FALSE.; RETURN
   ENDIF
  ENDDO
  IF( la < lb )THEN
    DO i = la+1,lb
      IF( blank > string_b%chars(i) )THEN
        s_ge_c = .TRUE.; RETURN
      ELSEIF( blank < string_b%chars(i) )THEN
        s_ge_c = .FALSE.; RETURN
     ENDIF
    ENDDO
  ELSEIF( la > lb )THEN
    DO i = lb+1,la
      IF( string_a%chars(i) > blank )THEN
        s_ge_c = .TRUE.; RETURN
      ELSEIF( string_a%chars(i) < blank )THEN
        s_ge_c = .FALSE.; RETURN
      ENDIF
    ENDDO
  ENDIF
  s_ge_c = .TRUE.
ENDFUNCTION s_ge_c
LOGICAL :: s_ge_c
INTEGER :: ls, la, lb
la = LEN(string_a); lb = LEN(string_b); ls = MIN(la, lb)
DO i = 1, ls
  IF( string_a%chars(i) > string_b(i:i) )THEN
    s_ge_c = .TRUE.; RETURN
  ELSEIF( string_a%chars(i) < string_b(i:i) )THEN
    s_ge_c = .FALSE.; RETURN
  ENDIF
ENDDO
IF( la < lb )THEN
  IF( blank > string_b(la+1:lb) )THEN
    s_ge_c = .TRUE.; RETURN
  ELSEIF( blank < string_b(la+1:lb) )THEN
    s_ge_c = .FALSE.; RETURN
  ENDIF
ELSEIF( la > lb )THEN
  DO i = lb+1, la
    IF( string_a%chars(i) > blank )THEN
      s_ge_c = .TRUE.; RETURN
    ELSEIF( string_a%chars(i) < blank )THEN
      s_ge_c = .FALSE.; RETURN
    ENDIF
  ENDDO
ENDIF
s_ge_c = .TRUE.
ENDFUNCTION s_ge_c

FUNCTION c_ge_s(string_a, string_b) ! character>=string
  CHARACTER(LEN=*) , INTENT(IN) :: string_a
  type(VARYING_STRING), INTENT(IN) :: string_b
  LOGICAL :: c_ge_s
  INTEGER :: ls, la, lb
  la = LEN(string_a); lb = LEN(string_b); ls = MIN(la, lb)
  DO i = 1, ls
    IF( string_a(i:i) > string_b%chars(i) )THEN
      c_ge_s = .TRUE.; RETURN
    ELSEIF( string_a(i:i) < string_b%chars(i) )THEN
      c_ge_s = .FALSE.; RETURN
    ENDIF
  ENDDO
ENDIF
s_ge_c = .TRUE.
ENDFUNCTION s_ge_c

FUNCTION s_gt_s(string_a, string_b) ! string>string
  type(VARYING_STRING), INTENT(IN) :: string_a, string_b
  LOGICAL :: s_gt_s
  INTEGER :: ls, la, lb
  la = LEN(string_a); lb = LEN(string_b); ls = MIN(la, lb)
  DO i = 1, ls
    IF( string_a%chars(i) > string_b%chars(i) )THEN
      s_gt_s = .TRUE.; RETURN
    ELSEIF( string_a%chars(i) < string_b%chars(i) )THEN
      s_gt_s = .FALSE.; RETURN
    ENDIF
  ENDDO
ENDIF
s_ge_c = .TRUE.
ENDFUNCTION s_ge_c

!------ Greater-than operators -----------------------------------------------!
FUNCTION s_gt_s(string_a, string_b) ! string>string
  type(VARYING_STRING), INTENT(IN) :: string_a, string_b
  LOGICAL :: s_gt_s
  INTEGER :: ls, la, lb
  la = LEN(string_a); lb = LEN(string_b); ls = MIN(la, lb)
  DO i = 1, ls
    IF( string_a%chars(i) > string_b%chars(i) )THEN
      s_gt_s = .TRUE.; RETURN
    ELSEIF( string_a%chars(i) < string_b%chars(i) )THEN
      s_gt_s = .FALSE.; RETURN
    ENDIF
  ENDDO
ENDIF
s_ge_c = .TRUE.
ENDFUNCTION s_ge_c
IF( la < lb )THEN
DO i = la+1,lb
IF( blank > string_b%chars(i) )THEN
   s_gt_s = .TRUE.; RETURN
ELSEIF( blank < string_b%chars(i) )THEN
   s_gt_s = .FALSE.; RETURN
ENDIF
ENDDO
ELSEIF( la > lb )THEN
DO i = lb+1,la
IF( string_a%chars(i) > blank )THEN
   s_gt_s = .TRUE.; RETURN
ELSEIF( string_a%chars(i) < blank )THEN
   s_gt_s = .FALSE.; RETURN
ENDIF
ENDDO
ENDIF
s_gt_s = .FALSE.
ENDFUNCTION s_gt_s

FUNCTION s_gt_c(string_a,string_b) ! string>character
   type(VARYING_STRING),INTENT(IN) :: string_a
   CHARACTER(LEN=*) ,INTENT(IN) :: string_b
   LOGICAL :: s_gt_c
   INTEGER :: ls,la,lb
   la = LEN(string_a); lb = LEN(string_b); ls = MIN(la,lb)
   DO i = 1,ls
      IF( string_a%chars(i) > string_b(i:i) )THEN
         s_gt_c = .TRUE.; RETURN
      ELSEIF( string_a%chars(i) < string_b(i:i) )THEN
         s_gt_c = .FALSE.; RETURN
      ENDIF
   ENDDO
   IF( la < lb )THEN
      IF( blank > string_b(la+1:lb) )THEN
         s_gt_c = .TRUE.; RETURN
      ELSEIF( blank < string_b(la+1:lb) )THEN
         s_gt_c = .FALSE.; RETURN
      ENDIF
   ELSEIF( la > lb )THEN
      DO i = lb+1,la
         IF( string_a%chars(i) > blank )THEN
            s_gt_c = .TRUE.; RETURN
         ELSEIF( string_a%chars(i) < blank )THEN
            s_gt_c = .FALSE.; RETURN
         ENDIF
      ENDDO
   ENDIF
   s_gt_c = .FALSE.
ENDFUNCTION s_gt_c

FUNCTION c_gt_s(string_a,string_b) ! character>string
   CHARACTER(LEN=*) ,INTENT(IN) :: string_a
   type(VARYING_STRING),INTENT(IN) :: string_b
   LOGICAL :: c_gt_s
   INTEGER :: ls,la,lb
   la = LEN(string_a); lb = LEN(string_b); ls = MIN(la,lb)
   DO i = 1,ls
      IF( string_a(i:i) > string_b%chars(i) )THEN
         c_gt_s = .TRUE.; RETURN
      ELSEIF( string_a(i:i) < string_b%chars(i) )THEN
         c_gt_s = .FALSE.; RETURN
      ENDIF
   ENDDO
   IF( la < lb )THEN
      DO i = la+1,lb
         IF( blank > string_b%chars(i) )THEN
            c_gt_s = .TRUE.; RETURN
         ELSEIF( blank < string_b%chars(i) )THEN
            c_gt_s = .FALSE.; RETURN
         ENDIF
      ENDDO
ELSEIF( la > lb )THEN
  IF( string_a(lb+1:la) > blank )THEN
    c_gt_s = .TRUE.; RETURN
  ELSEIF( string_a(lb+1:la) < blank )THEN
    c_gt_s = .FALSE.; RETURN
  ENDIF
ENDIF

ELSEIF( string_a(lb+1:la) > blank )THEN
  c_gt_s = .TRUE.; RETURN
ELSEIF( string_a(lb+1:la) < blank )THEN
  c_gt_s = .FALSE.; RETURN
ENDIF

c_gt_s = .FALSE.
ENDFUNCTION c_gt_s

!----- LLT procedures --------------------------------------------!
FUNCTION s_llt_s(string_a,string_b) ! string_a<string_b ISO-646 ordering
  type(VARYING_STRING),INTENT(IN) :: string_a,string_b
  LOGICAL :: s_llt_s
  ! Returns TRUE if string_a preceeds string_b in the ISO 646 collating
  ! sequence. Otherwise the result is FALSE. The result is FALSE if both
  ! string_a and string_b are zero length.
  INTEGER :: ls,la,lb
  la = LEN(string_a); lb = LEN(string_b); ls = MIN(la,lb)
  DO i = 1,ls
    IF( LLT(string_a%chars(i),string_b%chars(i)) )THEN
      s_llt_s = .TRUE.; RETURN
    ELSEIF( LGT(string_a%chars(i),string_b%chars(i)) )THEN
      s_llt_s = .FALSE.; RETURN
    ENDIF
  ENDDO
ENDIF

ELSEIF( la < lb )THEN
  DO i = la+1,lb
    IF( LLT(blank,string_b%chars(i)) )THEN
      s_llt_s = .TRUE.; RETURN
    ELSEIF( LGT(blank,string_b%chars(i)) )THEN
      s_llt_s = .FALSE.; RETURN
    ENDIF
  ENDDO
ENDIF

ELSEIF( la > lb )THEN
  DO i = lb+1,la
    IF( LLT(string_a%chars(i),blank) )THEN
      s_llt_s = .TRUE.; RETURN
    ELSEIF( LGT(string_a%chars(i),blank) )THEN
      s_llt_s = .FALSE.; RETURN
    ENDIF
  ENDDO
ENDIF

s_llt_s = .FALSE.
ENDFUNCTION s_llt_s

FUNCTION s_llt_c(string_a,string_b)
  type(VARYING_STRING),INTENT(IN) :: string_a
  CHARACTER(LEN=*) ,INTENT(IN) :: string_b
  LOGICAL :: s_llt_c
  INTEGER :: ls,la,lb
  la = LEN(string_a); lb = LEN(string_b); ls = MIN(la,lb)
  DO i = 1,ls
    IF( LLT(string_a%chars(i),string_b(i:i)) )THEN
      s_llt_c = .TRUE.; RETURN
    ELSEIF( LGT(string_a%chars(i),string_b(i:i)) )THEN
      s_llt_c = .FALSE.; RETURN
    ENDIF
  ENDDO
ENDIF

ELSEIF( la < lb )THEN
  IF( LLT(blank,string_b(la+1:lb)) )THEN
    s_llt_c = .TRUE.; RETURN
  ELSEIF( LGT(blank,string_b(la+1:lb)) )THEN
    s_llt_c = .FALSE.; RETURN
  ENDIF
ENDIF

ELSEIF( la > lb )THEN
  DO i = lb+1,la
    IF( LLT(string_a%chars(i),blank) )THEN
      s_llt_c = .TRUE.; RETURN
    ELSEIF( LGT(string_a%chars(i),blank) )THEN
      s_llt_c = .FALSE.; RETURN
    ENDIF
  ENDDO
ENDIF
FUNCTION c_llt_s(string_a,string_b) ! string_a,string_b ISO-646 ordering
CHARACTER(LEN=*) , INTENT(IN) :: string_a
type(VARYING_STRING), INTENT(IN) :: string_b
LOGICAL :: c_llt_s
INTEGER :: ls, la, lb
la = LEN(string_a); lb = LEN(string_b); ls = MIN(la,lb)
DO i = 1,ls
  IF( LLT(string_a(i:i),string_b%chars(i)) )THEN
    c_llt_s = .TRUE.; RETURN
  ELSEIF( LGT(string_a(i:i),string_b%chars(i)) )THEN
    c_llt_s = .FALSE.; RETURN
  ENDIF
ENDDO
IF( la < lb )THEN
  DO i = la+1,lb
    IF( LLT(blank,string_b%chars(i)) )THEN
      c_llt_s = .TRUE.; RETURN
    ELSEIF( LGT(blank,string_b%chars(i)) )THEN
      c_llt_s = .FALSE.; RETURN
    ENDIF
  ENDDO
ELSEIF( la > lb )THEN
  IF( LLT(string_a(lb+1:la),blank) )THEN
    c_llt_s = .TRUE.; RETURN
  ELSEIF( LGT(string_a(lb+1:la),blank) )THEN
    c_llt_s = .FALSE.; RETURN
  ENDIF
ENDIF
ENDIF

c_llt_s = .FALSE.
ENDFUNCTION c_llt_s

FUNCTION s_lle_s(string_a,string_b) ! string_a<=string_b ISO-646 ordering
CHARACTER(LEN=*) , INTENT(IN) :: string_a,string_b
LOGICAL :: s_lle_s
INTEGER :: ls, la, lb
la = LEN(string_a); lb = LEN(string_b); ls = MIN(la,lb)
DO i = 1,ls
  IF( LLT(string_a%chars(i),string_b%chars(i)) )THEN
    s_lle_s = .TRUE.; RETURN
  ELSEIF( LGT(string_a%chars(i),string_b%chars(i)) )THEN
    s_lle_s = .FALSE.; RETURN
  ENDIF
ENDDO
IF( la < lb )THEN
  DO i = la+1,lb
    IF( LLT(blank,string_b%chars(i)) )THEN
      s_lle_s = .TRUE.; RETURN
    ELSEIF( LGT(blank,string_b%chars(i)) )THEN
      s_lle_s = .FALSE.; RETURN
    ENDIF
  ENDDO
ELSEIF( la > lb )THEN
  IF( LLT(string_a%chars(i),blank) )THEN
    s_lle_s = .TRUE.; RETURN
  ELSEIF( LGT(string_a%chars(i),blank) )THEN
    s_lle_s = .FALSE.; RETURN
  ENDIF
ENDIF
ENDIF
s_lle_s = .TRUE.
ENDFUNCTION s_lle_s

FUNCTION s_lle_c(string_a,string_b) ! string_a<=string_b ISO-646 ordering
CHARACTER(LEN=*) , INTENT(IN) :: string_a,string_b
LOGICAL :: s_lle_c
INTEGER :: ls, la, lb
la = LEN(string_a); lb = LEN(string_b); ls = MIN(la,lb)
DO i = 1,ls
  IF( LLT(string_a%chars(i),string_b%chars(i)) )THEN
    s_lle_c = .TRUE.; RETURN
  ELSEIF( LGT(string_a%chars(i),string_b%chars(i)) )THEN
    s_lle_c = .FALSE.; RETURN
  ENDIF
ENDDO
IF( la < lb )THEN
  DO i = la+1,lb
    IF( LLT(blank,string_b%chars(i)) )THEN
      s_lle_c = .TRUE.; RETURN
    ELSEIF( LGT(blank,string_b%chars(i)) )THEN
      s_lle_c = .FALSE.; RETURN
    ENDIF
  ENDDO
ELSEIF( la > lb )THEN
  IF( LLT(string_a(lb+1:la),blank) )THEN
    s_lle_c = .TRUE.; RETURN
  ELSEIF( LGT(string_a(lb+1:la),blank) )THEN
    s_lle_c = .FALSE.; RETURN
  ENDIF
ENDIF
ENDIF
s_lle_c = .TRUE.
ENDFUNCTION s_lle_c
MODULE s_lle_c

 type(VARYING_STRING), INTENT(IN) :: string_a
 CHARACTER(LEN=*) , INTENT(IN) :: string_b
 LOGICAL :: s_lle_c
 INTEGER :: ls, la, lb

 la = LEN(string_a); lb = LEN(string_b); ls = MIN(la, lb)
 DO i = 1, ls
 IF ( LLT(string_a%chars(i), string_b(i:i)) ) THEN
     s_lle_c = .TRUE.; RETURN
 ELSEIF ( LGT(string_a%chars(i), string_b(i:i)) ) THEN
     s_lle_c = .FALSE.; RETURN
 ENDIF
 ENDDO

 IF ( la < lb ) THEN
     DO i = lb+1, la
     IF ( LLT(blank, string_b%chars(i)) ) THEN
         s_lle_c = .TRUE.; RETURN
     ELSEIF ( LGT(blank, string_b%chars(i)) ) THEN
         s_lle_c = .FALSE.; RETURN
     ENDIF
     ENDDO
 ENDIF

 s_lle_c = .TRUE.
ENDFUNCTION s_lle_c

FUNCTION c_lle_s(string_a,string_b) ! string_a<=string_b ISO-646 ordering

 CHARACTER(LEN=*) , INTENT(IN) :: string_a
 type(VARYING_STRING), INTENT(IN) :: string_b
 LOGICAL :: c_lle_s
 INTEGER :: ls, la, lb

 la = LEN(string_a); lb = LEN(string_b); ls = MIN(la, lb)
 DO i = 1, ls
 IF ( LLT(string_a(i:i), string_b%chars(i)) ) THEN
     c_lle_s = .TRUE.; RETURN
 ELSEIF ( LGT(string_a(i:i), string_b%chars(i)) ) THEN
     c_lle_s = .FALSE.; RETURN
 ENDIF
 ENDDO

 IF ( la < lb ) THEN
     DO i = la+1, lb
     IF ( LLT(blank, string_b%chars(i)) ) THEN
         c_lle_s = .TRUE.; RETURN
     ELSEIF ( LGT(blank, string_b%chars(i)) ) THEN
         c_lle_s = .FALSE.; RETURN
     ENDIF
     ENDDO
 ENDIF

 c_lle_s = .TRUE.
ENDFUNCTION c_lle_s

FUNCTION c_lle_s(string_a,string_b) ! string_a<=string_b ISO-646 ordering

 type(VARYING_STRING), INTENT(IN) :: string_a
 type(VARYING_STRING), INTENT(IN) :: string_b
 LOGICAL :: c_lle_s
 INTEGER :: ls, la, lb

 la = LEN(string_a); lb = LEN(string_b); ls = MIN(la, lb)
 DO i = 1, ls
 IF ( LLT(string_a%chars(i), string_b%chars(i)) ) THEN
     c_lle_s = .TRUE.; RETURN
 ELSEIF ( LGT(string_a%chars(i), string_b%chars(i)) ) THEN
     c_lle_s = .FALSE.; RETURN
 ENDIF
 ENDFUNCTION c_lle_s

FUNCTION s_lge_s(string_a,string_b) ! string_a>=string_b ISO-646 ordering

 type(VARYING_STRING), INTENT(IN) :: string_a
 type(VARYING_STRING), INTENT(IN) :: string_b
 LOGICAL :: s_lge_s
 INTEGER :: ls, la, lb

 la = LEN(string_a); lb = LEN(string_b); ls = MIN(la, lb)
 DO i = 1, ls
 IF ( LGT(string_a%chars(i), string_b%chars(i)) ) THEN
     s_lge_s = .TRUE.; RETURN
 ELSEIF ( LLT(blank, string_b%chars(i)) ) THEN
     s_lge_s = .FALSE.; RETURN
 ENDIF
 ENDFUNCTION s_lge_s

FUNCTION s_lge_s(string_a,string_b) ! string_a>=string_b ISO-646 ordering

 type(VARYING_STRING), INTENT(IN) :: string_a
 type(VARYING_STRING), INTENT(IN) :: string_b
 LOGICAL :: s_lge_s
 INTEGER :: ls, la, lb

 la = LEN(string_a); lb = LEN(string_b); ls = MIN(la, lb)
 DO i = 1, ls
 IF ( LGT(string_a%chars(i), string_b%chars(i)) ) THEN
     s_lge_s = .TRUE.; RETURN
 ENDIF
 ENDFUNCTION s_lge_s

ENDMODULE s_lle_c
ELSEIF( LLT(string_a%chars(i),string_b%chars(i)) )THEN
   s_lge_s = .FALSE.; RETURN
ENDIF
ENDDO

IF( la < lb )THEN
   DO i = la+1,lb
      IF( LGT(blank,string_b%chars(i)) )THEN
         s_lge_s = .TRUE.; RETURN
      ELSEIF( LLT(blank,string_b%chars(i)) )THEN
         s_lge_s = .FALSE.; RETURN
      ENDIF
   ENDDO
ENDIF

ELSEIF( la > lb )THEN
   DO i = lb+1,la
      IF( LGT(string_a%chars(i),blank) )THEN
         s_lge_s = .TRUE.; RETURN
      ELSEIF( LLT(string_a%chars(i),blank) )THEN
         s_lge_s = .FALSE.; RETURN
      ENDIF
   ENDDO
ENDIF

s_lge_s = .TRUE.
ENDFUNCTION s_lge_s

FUNCTION s_lge_c(string_a,string_b) ! string_a>=string_b ISO-646 ordering
   type(VARYING_STRING),INTENT(IN) :: string_a
   CHARACTER(LEN=*) ,INTENT(IN) :: string_b
   LOGICAL :: s_lge_c
   INTEGER :: ls,la,lb
   la = LEN(string_a); lb = LEN(string_b); ls = MIN(la,lb)
   DO i = 1,ls
      IF( LGT(string_a%chars(i),string_b(i:i)) )THEN
         s_lge_c = .TRUE.; RETURN
      ELSEIF( LLT(string_a%chars(i),string_b(i:i)) )THEN
         s_lge_c = .FALSE.; RETURN
      ENDIF
   ENDDO
   IF( la < lb )THEN
      IF( LGT(blank,string_b(la+1:lb)) )THEN
         s_lge_c = .TRUE.; RETURN
      ELSEIF( LLT(blank,string_b(la+1:lb)) )THEN
         s_lge_c = .FALSE.; RETURN
      ENDIF
   ENDIF
   ELSEIF( la > lb )THEN
      DO i = lb+1,la
         IF( LGT(string_a%chars(i),blank) )THEN
            s_lge_c = .TRUE.; RETURN
         ELSEIF( LLT(string_a%chars(i),blank) )THEN
            s_lge_c = .FALSE.; RETURN
         ENDIF
      ENDDO
   ENDIF
s_lge_c = .TRUE.
ENDFUNCTION s_lge_c

FUNCTION c_lge_s(string_a,string_b) ! string_a=>string_b ISO-646 ordering
   CHARACTER(LEN=*) ,INTENT(IN) :: string_a
   type(VARYING_STRING),INTENT(IN) :: string_b
   LOGICAL :: c_lge_s
   INTEGER :: ls,la,lb
   la = LEN(string_a); lb = LEN(string_b); ls = MIN(la,lb)
   DO i = 1,ls
      IF( LGT(string_a(i:i),string_b%chars(i)) )THEN
         c_lge_s = .TRUE.; RETURN
      ELSEIF( LLT(string_a(i:i),string_b%chars(i)) )THEN
         c_lge_s = .FALSE.; RETURN
      ENDIF
   ENDDO
   IF( la < lb )THEN
      DO i = la+1,lb
         IF( LGT(blank,string_b%chars(i)) )THEN
            c_lge_s = .TRUE.; RETURN
         ELSEIF( LLT(blank,string_b%chars(i)) )THEN
            c_lge_s = .FALSE.; RETURN
         ENDIF
      ENDDO
   ELSEIF( la > lb )THEN
      DO i = lb+1,la
         IF( LGT(string_a%chars(i),blank) )THEN
            c_lge_s = .TRUE.; RETURN
         ELSEIF( LLT(string_a%chars(i),blank) )THEN
            c_lge_s = .FALSE.; RETURN
         ENDIF
      ENDDO
   ENDIF
ELSEIF( LLT(blank,string_b%chars(i)) )THEN
  c_lge_s = .FALSE.; RETURN
ENDIF
ENDDO
ELSEIF( la > lb )THEN
  IF( LGT(string_a(lb+1:la),blank) )THEN
    c_lge_s = .TRUE.; RETURN
  ELSEIF( LLT(string_a(lb+1:la),blank) )THEN
    c_lge_s = .FALSE.; RETURN
  ENDIF
ENDIF
ENDIF
ENDFUNCTION c_lge_s

!----- LGT procedures -----------------------------------------------!

FUNCTION s_lgt_s(string_a,string_b) ! string_a>string_b ISO-646 ordering
  type(VARYING_STRING),INTENT(IN) :: string_a,string_b
  LOGICAL :: s_lgt_s
  ! Returns TRUE if string_a follows string_b in the ISO 646 collating sequence.
  ! Otherwise the result is FALSE. The result is FALSE if both string_a and
  ! string_b are zero length.
  INTEGER :: ls,la,lb
  la = LEN(string_a); lb = LEN(string_b); ls = MIN(la,lb)
  DO i = 1,ls
    IF( LGT(string_a%chars(i),string_b%chars(i)) )THEN
      s_lgt_s = .TRUE.; RETURN
    ELSEIF( LLT(string_a%chars(i),string_b%chars(i)) )THEN
      s_lgt_s = .FALSE.; RETURN
   ENDIF
  ENDDO
  IF( la < lb )THEN
    DO i = la+1,lb
      IF( LGT(blank,string_b%chars(i)) )THEN
        s_lgt_s = .TRUE.; RETURN
      ELSEIF( LLT(blank,string_b%chars(i)) )THEN
        s_lgt_s = .FALSE.; RETURN
     ENDIF
    ENDDO
  ELSEIF( la > lb )THEN
    DO i = lb+1,la
      IF( LGT(string_a%chars(i),blank) )THEN
        s_lgt_s = .TRUE.; RETURN
      ELSEIF( LLT(string_a%chars(i),blank) )THEN
        s_lgt_s = .FALSE.; RETURN
     ENDIF
    ENDDO
  ENDIF
ENDIF
ENDDO
ELSEIF( la > lb )THEN
  DO i = lb+1,la
    IF( LGT(string_a%chars(i),blank) )THEN
      s_lgt_s = .TRUE.; RETURN
    ELSEIF( LLT(string_a%chars(i),blank) )THEN
      s_lgt_s = .FALSE.; RETURN
   ENDIF
  ENDDO
ENDIF
s_lgt_s = .FALSE.
ENDFUNCTION s_lgt_s

FUNCTION s_lgt_c(string_a,string_b) ! string_a>string_b ISO-646 ordering
  type(VARYING_STRING),INTENT(IN) :: string_a
  CHARACTER(LEN=*) ,INTENT(IN) :: string_b
  LOGICAL :: s_lgt_c
  INTEGER :: ls,la,lb
  la = LEN(string_a); lb = LEN(string_b); ls = MIN(la,lb)
  DO i = 1,ls
    IF( LGT(string_a%chars(i),string_b(i:i)) )THEN
      s_lgt_c = .TRUE.; RETURN
    ELSEIF( LLT(string_a%chars(i),string_b(i:i)) )THEN
      s_lgt_c = .FALSE.; RETURN
   ENDIF
  ENDDO
  IF( la < lb )THEN
    IF( LGT(blank,string_b(la+1:lb)) )THEN
      s_lgt_c = .TRUE.; RETURN
    ELSEIF( LLT(blank,string_b(la+1:lb)) )THEN
      s_lgt_c = .FALSE.; RETURN
   ENDIF
  ELSEIF( la > lb )THEN
    DO i = lb+1,la
      IF( LGT(string_a%chars(i),blank) )THEN
        s_lgt_c = .TRUE.; RETURN
      ELSEIF( LLT(string_a%chars(i),blank) )THEN
        s_lgt_c = .FALSE.; RETURN
     ENDIF
  ENDIF
ENDIF
s_lgt_c = .FALSE.
ENDFUNCTION s_lgt_c
s_lgt_c = .TRUE.; RETURN
ELSEIF( LLT(string_a%chars(i),blank) )THEN
s_lgt_c = .FALSE.; RETURN
ENDIF
ENDDO
ENDIF
s_lgt_c = .FALSE.
ENDFUNCTION s_lgt_c

FUNCTION c_lgt_s(string_a,string_b) ! string_a>string_b ISO-646 ordering
CHARACTER(LEN=*) , INTENT(IN) :: string_a
! the string variable to be filled with
! characters read from the
! file connected to the default unit
INTEGER, INTENT(IN), OPTIONAL :: maxlen
! if present indicates the maximum
! number of characters that will be
! read from the file
INTEGER, INTENT(OUT), OPTIONAL :: iostat
! if present used to return the status
! of the data transfer
! if absent errors cause termination
! reads string from the default unit starting at next character in the file
! and terminating at the end of record or after maxlen characters.
CHARACTER(LEN=80) :: buffer
INTEGER :: ist,nch,toread,nb
IF(PRESENT(maxlen)) THEN
  toread=maxlen
ELSE
  toread=HUGE(1)
ENDIF
string = "" ! clears return string
DO ! repeatedly read buffer and add to string until EoR
  ! or maxlen reached
  IF(toread <= 0)EXIT
  nb=MIN(80,toread)
  READ(*,FMT=’(A)’,ADVANCE=’NO’,EOR=9999,SIZE=nch,IOSTAT=ist) buffer(1:nb)
  IF( ist /= 0 ) THEN
    IF(PRESENT(iostat)) THEN
      ! ! the string variable to be filled with
      ! characters read from the
      ! file connected to the default unit
      ! if present indicates the maximum
      ! number of characters that will be
      ! read from the file
      ! if present used to return the status
      ! of the data transfer
      ! if absent errors cause termination
      ! reads string from the default unit starting at next character in the file
      ! and terminating at the end of record or after maxlen characters.
      CHARACTER(LEN=80) :: buffer
      INTEGER :: ist,nch,toread,nb
      IF(PRESENT(maxlen)) THEN
        toread=maxlen
      ELSE
        toread=HUGE(1)
      ENDIF
      string = "" ! clears return string
      DO ! repeatedly read buffer and add to string until EoR
        ! or maxlen reached
        IF(toread <= 0)EXIT
        nb=MIN(80,toread)
        READ(*,FMT=’(A)’,ADVANCE=’NO’,EOR=9999,SIZE=nch,IOSTAT=ist) buffer(1:nb)
        IF( ist /= 0 ) THEN
          IF(PRESENT(iostat)) THEN
            ! ! the string variable to be filled with
            ! characters read from the
            ! file connected to the default unit
            ! if present indicates the maximum
            ! number of characters that will be
            ! read from the file
            ! if present used to return the status
            ! of the data transfer
            ! if absent errors cause termination
            ! reads string from the default unit starting at next character in the file
            ! and terminating at the end of record or after maxlen characters.
            CHARACTER(LEN=80) :: buffer
            INTEGER :: ist,nch,toread,nb
            IF(PRESENT(maxlen)) THEN
              toread=maxlen
            ELSE
              toread=HUGE(1)
            ENDIF
            string = "" ! clears return string
            DO ! repeatedly read buffer and add to string until EoR
              ! or maxlen reached
              IF(toread <= 0)EXIT
              nb=MIN(80,toread)
              READ(*,FMT=’(A)’,ADVANCE=’NO’,EOR=9999,SIZE=nch,IOSTAT=ist) buffer(1:nb)
              IF( ist /= 0 ) THEN
                IF(PRESENT(iostat)) THEN
                  ! ! the string variable to be filled with
                  ! characters read from the
                  ! file connected to the default unit
                  ! if present indicates the maximum
                  ! number of characters that will be
                  ! read from the file
                  ! if present used to return the status
                  ! of the data transfer
                  ! if absent errors cause termination
                  ! reads string from the default unit starting at next character in the file
                  ! and terminating at the end of record or after maxlen characters.
                  CHARACTER(LEN=80) :: buffer
                  INTEGER :: ist,nch,toread,nb
                  IF(PRESENT(maxlen)) THEN
                    toread=maxlen
                  ELSE
                    toread=HUGE(1)
                  ENDIF
                  string = "" ! clears return string
                  DO ! repeatedly read buffer and add to string until EoR
                    ! or maxlen reached
                    IF(toread <= 0)EXIT
                    nb=MIN(80,toread)
                    READ(*,FMT=’(A)’,ADVANCE=’NO’,EOR=9999,SIZE=nch,IOSTAT=ist) buffer(1:nb)
                    IF( ist /= 0 ) THEN
                      IF(PRESENT(iostat)) THEN
                        ! ! the string variable to be filled with
                        ! characters read from the
                        ! file connected to the default unit
                        ! if present indicates the maximum
                        ! number of characters that will be
                        ! read from the file
                        ! if present used to return the status
                        ! of the data transfer
                        ! if absent errors cause termination
                        ! reads string from the default unit starting at next character in the file
                        ! and terminating at the end of record or after maxlen characters.
SUBROUTINE get_u_eor(unit,string,maxlen,iostat)

INTEGER,INTENT(IN) :: unit
! identifies the input unit which must be
! connected for sequential formatted read

type(VARYING_STRING),INTENT(OUT) :: string
! the string variable to be filled with
! characters read from the
! file connected to the unit

INTEGER,INTENT(IN),OPTIONAL :: maxlen
! if present indicates the maximum
! number of characters that will be
! read from the file

INTEGER,INTENT(OUT),OPTIONAL :: iostat
! if present used to return the status
! of the data transfer
! if absent errors cause termination

!
! reads string from unit starting at next character in the file and
! terminating at the end of record or after maxlen characters.
CHARACTER(LEN=80) :: buffer

INTEGER :: ist,nch,toread,nb
IF(PRESENT(maxlen))THEN
  toread=maxlen
ELSE
  toread=HUGE(1)
ENDIF
string='' ! clears return string
DO ! repeatedly read buffer and add to string until EoR
  ! or maxlen reached
  IF(toread <= 0)EXIT
  nb=MIN(80,toread)
  READ(unit,FMT='(A)',ADVANCE='NO',EOR=9999,SIZE=nch,IOSTAT=ist) buffer(1:nb)
  IF( ist /= 0 ) THEN
    IF(PRESENT(iostat)) THEN
      iostat=ist
      RETURN
    ELSE
      WRITE(*,*) " Error No.",ist, &
      " during READ_STRING of varying string on UNIT ",unit
      STOP
    ENDIF
  ENDIF
  string = string //buffer(1:nb)
  toread = toread - nb
ENDDO
IF(PRESENT(iostat)) iostat = 0
RETURN
9999 string = string //buffer(1:nch)
IF(PRESENT(iostat)) iostat = ist
ENDSUBROUTINE get_u_eor

SUBROUTINE get_d_tset_s(string,set,maxlen,iostat)

! the string variable to be filled with
! characters read from the
! file connected to the default unit
![Natural text](https://example.com/natural_text.png)
! terminating at the end of record, occurrence of a character in set,
! or after reading maxlen characters.
CHARACTER :: buffer ! characters must be read one at a time to detect
! first terminator character in set
INTEGER :: ist,toread,lenset
ist=0
lenset = LEN(set)
IF (PRESENT(maxlen)) THEN
toread=maxlen
ELSE
  toread=HUGE(1)
ENDIF
string = "" ! clears return string
DO ! repeatedly read buffer and add to string until EoR
  ! or maxlen reached
  IF (toread <= 0) EXIT
  READ(unit,FMT='(A)',ADVANCE='NO',EOR=9999,IOSTAT=ist) buffer
  IF ( ist /= 0 ) THEN
    ist=ist
    RETURN
  ELSE
    WRITE(*,*) " Error No.",ist, &
    " during READ_STRING of varying string on unit ",unit
    STOP
  ENDIF
  ENDIF
! check for occurrence of set character in buffer
DO j = 1,lenset
  IF (buffer == set%chars(j)) GOTO 9999
ENDDO
string = string//buffer
  toread = toread - 1
ENDDO
IF (PRESENT(iostat)) iostat = ist
RETURN
9999 string = string//buffer
IF (PRESENT(iostat)) iostat = ist
ENDSUBROUTINE get_u_tset_s

SUBROUTINE get_d_tset_c(string,set,maxlen,iostat)
  type(VARYING_STRING),INTENT(OUT) :: string
  ! the string variable to be filled with
  ! characters read from the
  ! file connected to the default unit
  CHARACTER(LEN=*) ,INTENT(IN) :: set
  ! the set of characters which if found in
  ! the input terminate the read
  INTEGER,INTENT(IN),OPTIONAL :: maxlen
  ! if present indicates the maximum
  ! number of characters that will be
  ! read from the file
  INTEGER,INTENT(OUT),OPTIONAL :: iostat
  ! if present used to return the status
  ! of the data transfer
  ! if absent errors cause termination
  ! reads string from the default unit starting at next character in the file and
  ! terminating at the end of record, occurrence of a character in set,
  ! or after reading maxlen characters.
  CHARACTER :: buffer ! characters must be read one at a time to detect
  ! first terminator character in set
  INTEGER :: ist,toread,lenset
  ist=0
  lenset = LEN(set)
  IF (PRESENT(maxlen)) THEN
    toread=maxlen
  ELSE
    toread=HUGE(1)
 ENDIF
  string = "" ! clears return string
  DO ! repeatedly read buffer and add to string until EoR
    ! or maxlen reached
    READ(unit,FMT='(A)',ADVANCE='NO',EOR=9999,IOSTAT=ist) buffer
    IF ( ist /= 0 ) THEN
      ist=ist
      RETURN
    ELSE
      WRITE(*,*) " Error No.",ist, &
      " during READ_STRING of varying string on unit ",unit
      STOP
    ENDIF
  ENDIF
  IF (PRESENT(iostat)) iostat = ist
  RETURN
ENDSUBROUTINE get_d_tset_c
SUBROUTINE get_d_tset_c

INTEGER, INTENT(IN) :: unit

! identifies the input unit which must be connected for sequential formatted read

type(VARYING_STRING), INTENT(OUT) :: string

! the string variable to be filled with characters read from the file

INTEGER, INTENT(IN), OPTIONAL :: maxlen

! if present indicates the maximum number of characters that will be read from the file

INTEGER, INTENT(OUT), OPTIONAL :: iostat

! if present used to return the status of the data transfer

! if absent errors cause termination

! reads string from unit starting at next character in the file and
! terminating at the end of record, occurrence of a character in set,
! or after reading maxlen characters.

CHARACTER :: buffer ! characters must be read one at a time to detect first terminator character in set

INTEGER :: ist, toread, lenset

ist=0

lenset = LEN(set)

IF(PRESENT(maxlen)) THEN

toread=maxlen

ELSE

toread=HUGE(1)

ENDIF

string = "" ! clears return string

DO ! repeatedly read buffer and add to string until EoR

! or maxlen reached

IF(toread <= 0) EXIT

READ(unit, FMT='(A)', ADVANCE='NO', EOR=9999, IOSTAT=ist) buffer

IF( ist /= 0 ) THEN

iostat=ist

RETURN

ELSE

WRITE(*,*) " Error No.", ist, &
" during READ_STRING of varying string on default unit"

STOP

ENDIF

ENDIF

! check for occurrence of set character in buffer

DO j = 1, lenset

IF(buffer == set(j:j)) GOTO 9999

ENDDO

string = string//buffer

toread = toread - 1

ENDDO

IF(PRESENT(iostat)) iostat = ist

RETURN

9999 string = string//buffer

IF(PRESENT(iostat)) iostat = ist

ENDSUBROUTINE get_d_tset_c
! check for occurrence of set character in buffer
DO j = 1,lenset
   IF(buffer == set(j:j)) GOTO 9999
ENDDO
string = string//buffer
toread = toread - 1
ENDDO
IF(PRESENT(iostat)) iostat = ist
RETURN
9999 string = string//buffer
IF(PRESENT(iostat)) iostat = ist
ENDSUBROUTINE get_u_tset_c

!----- Output string procedures ----------------------------------------------!
SUBROUTINE put_d_s(string,iostat)
   type(VARYING_STRING),INTENT(IN) :: string
   INTEGER,INTENT(OUT),OPTIONAL :: iostat
   INTEGER :: ist
   WRITE(*,FMT='(A)',ADVANCE='NO',IOSTAT=ist) CHAR(string)
   IF( ist /= 0 )THEN
      IF(PRESENT(iostat)) THEN
         iostat = ist
         RETURN
      ELSE
         WRITE(*,*) " Error No.",ist, &
         " during WRITE_STRING of varying string on default unit"
         STOP
      ENDIF
   ENDIF
   IF(PRESENT(iostat)) iostat=0
ENDSUBROUTINE put_d_s
SUBROUTINE put_u_s(unit,string,iostat)
   INTEGER,INTENT(IN) :: unit
   type(VARYING_STRING),INTENT(IN) :: string
   INTEGER,INTENT(OUT),OPTIONAL :: iostat
   INTEGER :: ist
   WRITE(unit,FMT='(A)',ADVANCE='NO',IOSTAT=ist) CHAR(string)
   IF( ist /= 0 )THEN
      IF(PRESENT(iostat)) THEN
         iostat = ist
         RETURN
      ELSE
         WRITE(*,*) " Error No.",ist, &
         " during WRITE_STRING of varying string on UNIT ",unit
         STOP
      ENDIF
   ENDIF
   IF(PRESENT(iostat)) iostat=0
ENDSUBROUTINE put_u_s
SUBROUTINE put_d_c(string,iostat)
   CHARACTER(LEN=*) ,INTENT(IN) :: string
   INTEGER,INTENT(IN) :: unit
   WRITE(unit,FMT='(A)',ADVANCE='NO',IOSTAT=ist) CHAR(string)
   IF( ist /= 0 )THEN
      IF(PRESENT(iostat)) THEN
         iostat = ist
         RETURN
      ELSE
         WRITE(*,*) " Error No.",ist, &
         " during WRITE_STRING of varying string on UNIT ",unit
         STOP
      ENDIF
   ENDIF
   IF(PRESENT(iostat)) iostat=0
ENDSUBROUTINE put_d_c
! the next record if there is no
! current record

INTEGER, INTENT(OUT), OPTIONAL :: iostat
! if present used to return the status
! of the data transfer
! if absent errors cause termination

INTEGER :: ist
WRITE(*,FMT='(A)',ADVANCE='NO',IOSTAT=ist) string
IF( ist /= 0 ) THEN
  IF(PRESENT(iostat)) THEN
    iostat = ist
    RETURN
  ELSE
    WRITE(*,*) " Error No.", ist, &
    " during WRITE_STRING of character on default unit"
    STOP
  ENDIF
ENDIF
IF(PRESENT(iostat)) iostat=0
ENDSUBROUTINE put_d_c

SUBROUTINE put_u_c(unit,string,iostat)
  INTEGER, INTENT(IN) :: unit
  ! identifies the output unit which must
  ! be connected for sequential formatted
  ! write
  CHARACTER(LEN=*) , INTENT(IN) :: string
  ! the character variable to be appended to
  ! the current record or to the start of
  ! the next record if there is no
  ! current record
  INTEGER, INTENT(OUT), OPTIONAL :: iostat
  ! if present used to return the status
  ! of the data transfer
  ! if absent errors cause termination

  INTEGER :: ist
  WRITE(unit,FMT='(A)',ADVANCE='NO',IOSTAT=ist) string
  IF( ist /= 0 ) THEN
    IF(PRESENT(iostat)) THEN
      iostat = ist
      RETURN
    ELSE
      WRITE(*,*) " Error No.", ist, &
      " during WRITE_STRING of character on UNIT ",unit
      STOP
    ENDIF
  ELSE
    IF(PRESENT(iostat)) iostat=0
  ENDIF
ENDSUBROUTINE put_u_c

SUBROUTINE putline_d_s(string,iostat)
  type(VARYING_STRING), INTENT(IN) :: string
  ! the string variable to be appended to
  ! the current record or to the start of
  ! the next record if there is no
  ! current record
  INTEGER, INTENT(OUT), OPTIONAL :: iostat
  ! if present used to return the status
  ! of the data transfer
  ! if absent errors cause termination

  INTEGER :: ist
  WRITE(*,FMT='(A,/)',ADVANCE='NO',IOSTAT=ist) CHAR(string)
  IF( ist /= 0 ) THEN
    IF(PRESENT(iostat)) THEN
      iostat = ist; RETURN
    ELSE
      WRITE(*,*) " Error No.", ist, &
      " during WRITE_LINE of varying string on default unit"
STOP
ENDIF
IF(PRESENT(iostat)) iostat=0
ENDSUBROUTINE putline_d_s

SUBROUTINE putline_u_s(unit,string,iostat)
  INTEGER,INTENT(IN) :: unit
  ! identifies the output unit which must
  ! be connected for sequential formatted
  ! write
  type(VARYING_STRING),INTENT(IN) :: string
  ! the string variable to be appended to
  ! the current record or to the start of
  ! the next record if there is no
  ! current record
  INTEGER,INTENT(OUT),OPTIONAL :: iostat
  ! if present used to return the status
  ! of the data transfer
  ! if absent errors cause termination
  INTEGER :: ist
  WRITE(unit,FMT='(A,/)',ADVANCE='NO',IOSTAT=ist) CHAR(string)
  IF( ist /= 0 )THEN
    IF(PRESENT(iostat))THEN
      iostat = ist; RETURN
    ELSE
      WRITE(*,*) " Error No.",ist, 
      " during WRITE_LINE of varying string on UNIT",unit
      STOP
    ENDIF
  ENDIF
  IF(PRESENT(iostat)) iostat=0
ENDSUBROUTINE putline_u_s

SUBROUTINE putline_d_c(string,iostat)
  CHARACTER(LEN=*)) :: string
  ! the character variable to be appended to
  ! the current record or to the start of
  ! the next record if there is no
  ! current record
  ! uses the default unit
  INTEGER,INTENT(OUT),OPTIONAL :: iostat
  ! if present used to return the status
  ! of the data transfer
  ! if absent errors cause termination
  INTEGER :: ist
  WRITE(*,FMT='(A,/)',ADVANCE='NO',IOSTAT=ist) string
  IF(PRESENT(iostat))THEN
    iostat = ist
    RETURN
  ELSEIF( ist /= 0 )THEN
    WRITE(*,*) " Error No.",ist, 
    " during WRITE_LINE of character on default unit"
    STOP
  ENDIF
ENDSUBROUTINE putline_d_c

SUBROUTINE putline_u_c(unit,string,iostat)
  INTEGER,INTENT(IN) :: unit
  ! identifies the output unit which must
  ! be connected for sequential formatted
  ! write
  CHARACTER(LEN=*)) :: string
  ! the character variable to be appended to
  ! the current record or to the start of
  ! the next record if there is no
  ! current record
  INTEGER,INTENT(OUT),OPTIONAL :: iostat
  ! if present used to return the status
  ! of the data transfer
  ! if absent errors cause termination
  INTEGER :: ist
  WRITE(*,FMT='(A,/)',ADVANCE='NO',IOSTAT=ist) string
  IF(PRESENT(iostat))THEN
    iostat = ist
    RETURN
  ELSEIF( ist /= 0 )THEN
    WRITE(*,*) " Error No.",ist, 
    " during WRITE_LINE of character on default unit"
    STOP
  ENDIF
ENDSUBROUTINE putline_u_c
INTEGER, INTENT(OUT), OPTIONAL :: iostat
  ! if present used to return the status
  ! of the data transfer
  ! if absent errors cause termination

! appends the string to the current record and then ends the record
! leaves the file positioned after the record just completed which then
! becomes the previous and last record in the file.

INTEGER :: ist

WRITE(unit,FMT='(A,/)',ADVANCE='NO',IOSTAT=ist) string
IF(PRESENT(iostat)) THEN
  iostat = ist
  RETURN
ELSEIF( ist /= 0 ) THEN
  WRITE(*,*) " Error No.",ist, &
  " during WRITE_LINE of character on UNIT",unit
  STOP
ENDIF
ENDSUBROUTINE putline_u_c

!----- Insert procedures ----------------------------------------------------!

FUNCTION insert_ss(string,start,substring)
  type(VARYING_STRING) :: insert_ss
  type(VARYING_STRING), INTENT(IN) :: string
  INTEGER, INTENT(IN) :: start
  type(VARYING_STRING), INTENT(IN) :: substring

  ! calculates result string by inserting the substring into string
  ! beginning at position start pushing the remainder of the string
  ! to the right and enlarging it if absent errors cause termination.
  ! if start is greater than LEN(string) the substring is simply appended
  ! to string by concatenation. if start is less than 1
  ! substring is inserted before string, i.e. start is treated as if it were 1

  CHARACTER, POINTER, DIMENSION(:) :: work
  INTEGER :: ip, is, lsub, ls
  lsub = LEN(substring); ls = LEN(string)
  is = MAX(start,1)
  ip = MIN(ls+1,is)
  ALLOCATE(work(1:lsub+ls))
  work(1:ip-1) = string%chars(1:ip-1)
  work(ip:ip+lsub-1) = substring%chars
  work(ip+lsub:lsub+ls) = string%chars(ip:ls)
  insert_ss%chars => work
ENDFUNCTION insert_ss

FUNCTION insert_sc(string,start,substring)
  type(VARYING_STRING) :: insert_sc
  type(VARYING_STRING), INTENT(IN) :: string
  INTEGER, INTENT(IN) :: start
  CHARACTER(LEN=*) :: substring

  ! calculates result string by inserting the substring into string
  ! beginning at position start pushing the remainder of the string
  ! to the right and enlarging it accordingly,
  ! if start is greater than LEN(string) the substring is simply appended
  ! to string by concatenation. if start is less than 1
  ! substring is inserted before string, i.e. start is treated as if it were 1

  CHARACTER, POINTER, DIMENSION(:) :: work
  INTEGER :: ip, is, lsub, ls
  lsub = LEN(substring); ls = LEN(string)
  is = MAX(start,1)
  ip = MIN(ls+1,is)
  ALLOCATE(work(1:lsub+ls))
  work(1:ip-1) = string%chars(1:ip-1)
  DO i = 1,lsub
    work(ip-1+i) = substring(i:i)
  ENDDO
  work(ip+lsub:lsub+ls) = string%chars(ip:ls)
  insert_sc%chars => work
ENDFUNCTION insert_sc

FUNCTION insert_cs(string,start,substring)
  type(VARYING_STRING) :: insert_cs
  CHARACTER(LEN=*) :: string

  ! calculates result string by inserting the substring into string
  ! beginning at position start pushing the remainder of the string
  ! to the right and enlarging it accordingly,
  ! if start is greater than LEN(string) the substring is simply appended
  ! to string by concatenation. if start is less than 1
  ! substring is inserted before string, i.e. start is treated as if it were 1

  CHARACTER, POINTER, DIMENSION(:) :: work
  INTEGER :: ip, is, lsub, ls
  lsub = LEN(substring); ls = LEN(string)
  is = MAX(start,1)
  ip = MIN(ls+1,is)
  ALLOCATE(work(1:lsub+ls))
  work(1:ip-1) = string%chars(1:ip-1)
  DO i = 1,lsub
    work(ip+i) = substring(i:i)
  ENDDO
  work(ip+lsub:lsub+ls) = string%chars(ip:ls)
  insert_cs%chars => work
ENDFUNCTION insert_cs
INTEGER, INTENT(IN) :: start

TYPE(VARYING_STRING), INTENT(IN) :: substring

! calculates result string by inserting the substring into string 
! beginning at position start pushing the remainder of the string 
! to the right and enlarging it accordingly.

! if start is greater than LEN(string) the substring is simply appended
! to string by concatenation. if start is less than 1

! substring is inserted before string, i.e. start is treated as if it were 1

CHARACTER, POINTER, DIMENSION(:) :: work

INTEGER :: ip, is, lsub, ls

lsub = LEN(substring); ls = LEN(string)

is = MAX(start, 1)

ip = MIN(ls+1, is)

ALLOCATE(work(1:lsub+ls))

DO i=1,ip-1
    work(i) = string(i:i)
ENDDO

DO i = 1,lsub
    work(ip-1+i) = substring(i:i)
ENDDO

DO i=ip,ls
    work(i+lsub) = string(i:i)
ENDDO

insert_cs%chars => work

ENDFUNCTION insert_cs

FUNCTION insert_cs(string,start,substring)

TYPE(VARYING_STRING) :: insert_cs

CHARACTER(LEN=*) :: string

INTEGER, INTENT(IN) :: start

CHARACTER(LEN=*) :: substring

! calculates result string by inserting the substring into string 
! beginning at position start pushing the remainder of the string 
! to the right and enlarging it accordingly,

! if start is greater than LEN(string) the substring is simply appended
! to string by concatenation. if start is less than 1

! substring is inserted before string, i.e. start is treated as if it were 1

CHARACTER, POINTER, DIMENSION(:) :: work

INTEGER :: ip, is, lsub, ls

lsub = LEN(substring); ls = LEN(string)

is = MAX(start, 1)

ip = MIN(ls+1, is)

ALLOCATE(work(1:lsub+ls))

DO i=1,ip-1
    work(i) = string(i:i)
ENDDO

DO i = 1,lsub
    work(ip-1+i) = substring(i:i)
ENDDO

DO i=ip,ls
    work(i+lsub) = string(i:i)
ENDDO

endcs%chars => work

ENDFUNCTION insert_cs

FUNCTION replace_ss(string,start,substring)

TYPE(VARYING_STRING) :: replace_ss

TYPE(VARYING_STRING), INTENT(IN) :: string

INTEGER, INTENT(IN) :: start

TYPE(VARYING_STRING), INTENT(IN) :: substring

! calculates the result string by the following actions:
! inserts the substring into string beginning at position 
! start replacing the following LEN(substring) characters of the string 
! and enlarging string if necessary, if start is greater than LEN(string)

! substring is simply appended to string by concatenation. If start is less 
! than 1, substring replaces characters in string starting at 1

CHARACTER, POINTER, DIMENSION(:) :: work

INTEGER :: ip, is, nw, lsub, ls

lsub = LEN(substring); ls = LEN(string)

is = MAX(start, 1)

ip = MIN(ls+1, is)

ALLOCATE(work(1:lsub+ls))

DO i=1,ip-1
    work(i) = string(i:i)
ENDDO

DO i = 1,lsub
    work(ip-1+i) = substring(i:i)
ENDDO

ENDDO

DO i = ip, ls
    work(i+lsub) = string(i:i)
ENDDO

ENDDO

insert_cs%chars => work

ENDFUNCTION insert_cs

!---------------- Replace procedures --------------------------!
FUNCTION replace_ss_sf(string, start, finish, substring)
  type(VARYING_STRING) :: replace_ss_sf
  type(VARYING_STRING), INTENT(IN) :: string
  INTEGER, INTENT(IN) :: start, finish
  type(VARYING_STRING), INTENT(IN) :: substring
  ! calculates the result string by the following actions:
  ! inserts the characters from substring into string beginning at position
  ! and enlarging if necessary. If start is greater than LEN(string) substring is
  ! simply appended to string by concatenation. If start is less than 1, substring
  ! replaces characters in string starting at 1
  CHARACTER, POINTER, DIMENSION(:) :: work
  INTEGER :: ip, is, if, nw, lsub, ls
  lsub = LEN(substring); ls = LEN(string)
  is = MAX(start, 1)
  ip = MIN(ls+1, is)
  if = MAX(ip-1, MIN(finish, ls))
  nw = lsub + ls - if+ip-1
  ALLOCATE(work(1:nw))
  work(1:ip-1) = string%chars(1:ip-1)
  DO i = 1, lsub
    work(ip-1+i) = substring(i:i)
  ENDDO
  work(ip+lsub:nw) = string%chars(ip+lsub:ls)
  replace_ss_sf%chars => work
ENDFUNCTION replace_ss_sf

FUNCTION replace_sc(string, start, substring)
  type(VARYING_STRING) :: replace_sc
  type(VARYING_STRING), INTENT(IN) :: string
  INTEGER, INTENT(IN) :: start
  CHARACTER(LEN=*) :: substring
  ! calculates the result string by the following actions:
  ! inserts the characters from substring into string beginning at position
  ! and enlarging if necessary. If start is greater than LEN(string) substring is
  ! simply appended to string by concatenation. If start is less than
  ! LEN(string), finish = LEN(string) is used
  ! If finish is less than start, substring is inserted before start
  CHARACTER, POINTER, DIMENSION(:) :: work
  INTEGER :: ip, is, lw, lsub, ls
  lsub = LEN(substring); ls = LEN(string)
  is = MAX(start, 1)
  ip = MIN(ls+1, is)
  lw = MAX(ip, ls)
  ALLOCATE(work(1:lw))
  work(1:ip-1) = string%chars(1:ip-1)
  work(ip+1:ip+lsub-1) = substring%chars(ip:ip+lsub-1)
  work(ip+lsub:nw) = string%chars(ip+lsub:ls)
  replace_sc%chars => work
ENDFUNCTION replace_sc

FUNCTION replace_sc_sf(string, start, finish, substring)
  type(VARYING_STRING) :: replace_sc_sf
  type(VARYING_STRING), INTENT(IN) :: string
  INTEGER, INTENT(IN) :: start, finish
  CHARACTER(LEN=*) :: substring
  ! calculates the result string by the following actions:
  ! inserts the substring into string beginning at position
  ! and enlarging or shrinking the string if necessary.
  ! If start is greater than LEN(string) substring is simply appended to string
  ! by concatenation. If start is less than 1, start = 1 is used
  ! If finish is greater than LEN(string), finish = LEN(string) is used
  ! If finish is less than start, substring is inserted before start
  CHARACTER, POINTER, DIMENSION(:) :: work
  INTEGER :: ip, is, if, nw, lsub, ls
  lsub = LEN(substring); ls = LEN(string)
  is = MAX(start, 1)
  ip = MIN(ls+1, is)
  if = MAX(ip-1, MIN(finish, ls))
  nw = lsub + ls - if+ip-1
  ALLOCATE(work(1:nw))
  work(1:ip-1) = string%chars(1:ip-1)
  work(ip:ip+lsub-1) = substring%chars(ip:ip+lsub-1)
  work(ip+lsub:nw) = string%chars(ip+lsub:ls)
  replace_sc_sf%chars => work
ENDFUNCTION replace_sc_sf
ISO/IEC 1539-2 : 1993 (E)  

FUNCTION replace_sc_sf(string,start,finish,substring)  
  type(VARYING_STRING) :: replace_sc_sf  
  CHARACTER(LEN=*) , INTENT(IN) :: string  
  INTEGER, INTENT(IN) :: start, finish  
  type(VARYING_STRING), INTENT(IN) :: substring  
  ! calculates the result string by the following actions:  
  ! inserts the substring into string beginning at position  
  ! start replacing the following LEN(substring) characters of the string  
  ! and enlarging string if necessary. if start is greater than LEN(string)  
  ! substring is simply appended to string by concatenation. If start is less  
  ! than 1, substring replaces characters in string starting at 1  
  CHARACTER, POINTER, DIMENSION(:) :: work  
  INTEGER :: ip,is,if,nw,lsub,ls  
  lsub = LEN(substring); ls = LEN(string)  
  is = MAX(start,1)  
  ip = MIN(ls+1,is)  
  if = MAX(ip-1,MIN(finish,ls))  
  nw = lsub + ls - if+ip-1  
  ALLOCATE(work(1:nw))  
  work(1:ip-1) = string%chars(1:ip-1)  
  DO i = 1,lsub  
    work(ip-1+i) = substring(i:i)  
  ENDDO  
  work(ip+lsub:nw) = string%chars(if+1:ls)  
  replace_sc_sf%chars => work  
ENDFUNCTION replace_sc_sf  

FUNCTION replace_cs(string,start,substring)  
  type(VARYING_STRING) :: replace_cs  
  CHARACTER(LEN=*) , INTENT(IN) :: string  
  INTEGER, INTENT(IN) :: start  
  type(VARYING_STRING), INTENT(IN) :: substring  
  ! calculates the result string by the following actions:  
  ! inserts the substring into string beginning at position  
  ! start replacing the following LEN(substring) characters of the string  
  ! and enlarging string if necessary. if start is greater than LEN(string)  
  ! substring is simply appended to string by concatenation. If start is less  
  ! than 1, substring replaces characters in string starting at 1  
  CHARACTER, POINTER, DIMENSION(:) :: work  
  INTEGER :: ip,is,if,nw,lsub,ls  
  lsub = LEN(substring); ls = LEN(string)  
  is = MAX(start,1)  
  ip = MIN(ls+1,is)  
  nw = MAX(ls,ip+lsub-1)  
  ALLOCATE(work(1:nw))  
  DO i=1,ip-1  
    work(i) = string(i:i)  
  ENDDO  
  work(ip:ip+lsub-1) = substring%chars  
  DO i=ip+lsub,nw  
    work(i) = string(i:i)  
  ENDDO  
  replace_cs%chars => work  
ENDFUNCTION replace_cs  

FUNCTION replace_cs_sf(string,start,finish,substring)  
  type(VARYING_STRING) :: replace_cs_sf  
  CHARACTER(LEN=*) , INTENT(IN) :: string  
  INTEGER, INTENT(IN) :: start, finish  
  type(VARYING_STRING), INTENT(IN) :: substring  
  ! calculates the result string by the following actions:  
  ! inserts the substring into string beginning at position  
  ! start replacing the following finish-start+1 characters of the string  
  ! and enlarging or shrinking the string if necessary.  
  ! If start is greater than LEN(string) substring is simply appended to string  
  ! by concatenation. If start is less than 1, start = 1 is used  
  ! If finish is greater than LEN(string), finish = LEN(string) is used  
  ! If finish is less than start, substring is inserted before start  
  CHARACTER, POINTER, DIMENSION(:) :: work  
  INTEGER :: ip,is,if,nw,lsub,ls  
  lsub = LEN(substring); ls = LEN(string)  
  is = MAX(start,1)  
  ip = MIN(ls+1,is)  
  if = MAX(ip-1,MIN(finish,ls))  
  nw = lsub + ls - if+ip-1  
  ALLOCATE(work(1:nw))  
  DO i=1,ip-1  
    work(i) = string(i:i)  
  ENDDO  
  work(ip:ip+lsub-1) = substring%chars  
  DO i=ip+1:ip+lsub  
    work(i) = string(if+i:if+i)  
  ENDDO  
  replace_cs_sf%chars => work  
ENDFUNCTION replace_cs_sf
FUNCTION replace_cs_sf(string, start, substring) 
CHARACTER(Len=*) :: replace_cs_sf 
INTEGER, INTENT(IN) :: string 
INTEGER, INTENT(IN) :: start 
CHARACTER(Len=*) :: substring 
ENDFUNCTION replace_cs_sf

FUNCTION replace_cc(string, start, substring) 
CHARACTER(Len=*) :: replace_cc 
INTEGER, INTENT(IN) :: string 
INTEGER, INTENT(IN) :: start 
CHARACTER(Len=*) :: substring 
ENDFUNCTION replace_cc

FUNCTION replace_cc_sf(string, start, finish, substring) 
CHARACTER(Len=*) :: replace_cc_sf 
INTEGER, INTENT(IN) :: string 
INTEGER, INTENT(IN) :: start 
INTEGER, INTENT(IN) :: finish 
CHARACTER(Len=*) :: substring 
ENDFUNCTION replace_cc_sf

FUNCTION replace_sss(string, target, substring, every, back) 
CHARACTER(Len=*) :: replace_sss 
CHARACTER(Len=*) :: string, target, substring 
LOGICAL, INTENT(IN), OPTIONAL :: every, back 
ENDFUNCTION replace_sss
SUBROUTINE replace_sss(string,target,substring,every,back)

! substring. if back present with value true search is backward otherwise
! search is done forward. if every present with value true all occurrences
! of target in string are replaced, otherwise only the first found is
! replaced. if target is not found the result is the same as string.

LOGICAL :: dir_switch, rep_search
CHARACTER,DIMENSION(:),POINTER :: work,temp
INTEGER :: ls,lt,lsub,ipos,ipow
ls = LEN(string); lt = LEN(target); lsub = LEN(substring)

IF(lt==0)THEN
  WRITE(*,*) " Zero length target in REPLACE"
  STOP
ENDIF

ALLOCATE(work(1:ls)); work = string%chars
IF( PRESENT(back) )THEN
  dir_switch = back
ELSE
  dir_switch = .FALSE.
ENDIF
IF( PRESENT(every) )THEN
  rep_search = every
ELSE
  rep_search = .FALSE.
ENDIF

IF( dir_switch )THEN ! backwards search
  ipos = ls-lt+1
  DO
    IF( ipos < 1 )EXIT ! search past start of string
    ! test for occurrence of target in string at this position
    IF( .ALL.(string%chars(ipos:ipos+lt-1) == target%chars) )THEN
      ! match found allocate space for string with this occurrence of
      ! target replaced by substring
      ALLOCATE(temp(1:SIZE(work)+lsub-lt))
      ! copy work into temp replacing this occurrence of target by
      ! substring
      temp(1:ipos-1) = work(1:ipos-1)
      temp(ipos:ipos+lsub-1) = substring%chars
      temp(ipos+lsub:) = work(ipos+lt:)
      work => temp ! make new version of work point at the temp space
      IF(.NOT.rep_search)EXIT ! exit if only first replacement wanted
      ! move search and replacement positions over the effected positions
      ipos = ipos-lt+1
    ENDDO
ELSE ! forward search
  ipos = 1; ipow = 1
  DO
    IF( ipos > ls-lt+1 )EXIT ! search past end of string
    ! test for occurrence of target in string at this position
    IF( .ALL.(string%chars(ipos:ipos+lt-1) == target%chars) )THEN
      ! match found allocate space for string with this occurrence of
      ! target replaced by substring
      ALLOCATE(temp(1:SIZE(work)+lsub-lt))
      ! copy work into temp replacing this occurrence of target by
      ! substring
      temp(1:ipow-1) = work(1:ipow-1)
      temp(ipow:ipow+lsub-1) = substring%chars
      temp(ipow+lsub:) = work(ipow+lt:)
      work => temp ! make new version of work point at the temp space
      IF(.NOT.rep_search)EXIT ! exit if only first replacement wanted
      ! move search and replacement positions over the effected positions
      ipos = ipos+lt-1; ipow = ipow+lsub-1
    ENDDO
  ELSE ! forward search
ENDIF
ELSE ! forward search
ENDIF
ENDfunction replace_sss

FUNCTION replace_ssc(string,target,substring,every,back)

ENDFUNCTION replace_ssc

FUNCTION replace_ssc(string,target,substring,every,back)

ENDFUNCTION replace_ssc

FUNCTION replace_ssc(string,target,substring,every,back)

ENDFUNCTION replace_ssc
CD version 2 ISO/IEC 1539-2 : 1993 (E)

CHARACTER(LEN=*) :: substring
LOGICAL, INTENT(IN), OPTIONAL :: every, back

! calculates the result string by the following actions:
! searches for occurrences of target in string, and replaces these with
! substring. if back present with value true search is backward otherwise
! search is done forward. if every present with value true all occurrences
! of target in string are replaced, otherwise only the first found is
! replaced. if target is not found the result is the same as string.

LOGICAL :: dir_switch, rep_search
CHARACTER, DIMENSION(:,), POINTER :: work, temp
INTEGER :: ls, lt, lsub, ipos, ipow

ls = LEN(string); lt = LEN(target); lsub = LEN(substring)
IF(lt==0) THEN
  WRITE(*,*) " Zero length target in REPLACE"
  STOP
ENDIF
ALLOCATE(work(1:ls)); work = string%chars
IF(PRESENT(back)) THEN
  dir_switch = back
ELSE
  dir_switch = .FALSE.
ENDIF
IF(PRESENT(every)) THEN
  rep_search = every
ELSE
  rep_search = .FALSE.
ENDIF
IF(dir_switch) THEN ! backwards search
  ipos = ls-lt+1
  DO
    IF( ipos < 1 ) EXIT ! search past start of string
    ! test for occurrence of target in string at this position
    IF( ALL(string%chars(ipos:ipos+lt-1) == target%chars) ) THEN
      ! match found allocate space for string with this occurrence of
      ! target replaced by substring
      ALLOCATE(temp(1:SIZE(work)+lsub-lt))
      ! copy work into temp replacing this occurrence of target by
      ! substring
      temp(1:ipos-1) = work(1:ipos-1)
      DO i=1,lsub
        temp(i+ipos-1) = substring(i:i)
      ENDDO
      temp(ipos+lsub:) = work(ipos+lt:)
      work => temp ! make new version of work point at the temp space
      IF(.NOT.rep_search) EXIT ! exit if only first replacement wanted
      ! move search and replacement positions over the effected positions
      ipos = ipos-lt+1
      ENDDO
  ENDMIV
  ipos=ipos-1
  ENDDO
ELSE ! forward search
  ipos = 1; ipow = 1
  DO
    IF( ipos > ls-lt+1 ) EXIT ! search past end of string
    ! test for occurrence of target in string at this position
    IF( ALL(string%chars(ipos:ipos+lt-1) == target%chars) ) THEN
      ! match found allocate space for string with this occurrence of
      ! target replaced by substring
      ALLOCATE(temp(1:SIZE(work)+lsub-lt))
      ! copy work into temp replacing this occurrence of target by
      ! substring
      temp(1:ipow-1) = work(1:ipow-1)
      DO i=1,lsub
        temp(i+ipow-1) = substring(i:i)
      ENDDO
      temp(ipow+lsub:) = work(ipow+lt:)
      work => temp ! make new version of work point at the temp space
      IF(.NOT.rep_search) EXIT ! exit if only first replacement wanted
      ! move search and replacement positions over the effected positions
      ipos = ipos+lt-1; ipow = ipow+lsub-1
      ENDDO
  ENDMIV
  ipos=ipos+1; ipow=ipow+1
ENDFUNCTION replace_scc

FUNCTION replace_scs(string, target, substring, every, back)
  type(VARYING_STRING) :: replace_scs
  type(VARYING_STRING), INTENT(IN) :: string, substring
  CHARACTER(LEN=*) , INTENT(IN) :: target
  LOGICAL, INTENT(IN), OPTIONAL :: every, back

  ! calculates the result string by the following actions:
  ! searches for occurrences of target in string, and replaces these with
  ! substring. if back present with value true search is backward otherwise
  ! search is done forward. if every present with value true all accurences
  ! of target in string are replaced, otherwise only the first found is
  ! replaced. if target is not found the result is the same as string.

  LOGICAL :: dir_switch, rep_search
  CHARACTER, DIMENSION(:,), POINTER :: work, temp, tget
  INTEGER :: ls, lt, lsub, ipos, ipow

  ls = LEN(string); lt = LEN(target); lsub = LEN(substring)

  IF(lt==0) THEN
    WRITE(*,*) " Zero length target in REPLACE"
    STOP
  ENDIF

  ALLOCATE(work(1:ls)); work = string%chars
  ALLOCATE(tget(1:lt))
  DO i=1,lt
    tget(i) = target(i:i)
  ENDDO

  IF( PRESENT(back) ) THEN
    dir_switch = back
  ELSE
    dir_switch = .FALSE.
  ENDIF
  IF( PRESENT(every) ) THEN
    rep_search = every
  ELSE
    rep_search = .FALSE.
  ENDIF

  IF( dir_switch ) THEN ! backwards search
    ipos = ls-lt+1
    DO
      IF( ipos < 1 ) EXIT ! search past start of string
      IF( ALL(string%chars(ipos:ipos+lt-1) == tget) ) THEN
        ! match found allocate space for string with this occurrence of
        ! target replaced by substring
        ALLOCATE(temp(1:SIZE(work)+lsub-lt))
        ! copy work into temp replacing this occurrence of target by
        ! substring
        temp(1:ipos-1) = work(1:ipos-1)
        temp(ipos:ipos+lsub-1) = substring%chars
        work => temp ! make new version of work point at the temp space
        IF(.NOT.rep_search)EXIT ! exit if only first replacement wanted
        ! move search and replacement positions over the effected positions
        ipos = ipos-lt+1
      ENDIF
      ipos=ipos-1
    ENDDO
  ELSE ! forward search
    ipos = 1; ipow = 1
    DO
      IF( ipos > ls-lt+1 ) EXIT ! search past end of string
      IF( ALL(string%chars(ipos:ipos+lt-1) == tget) ) THEN
        ! match found allocate space for string with this occurrence of
        ! target replaced by substring
        ALLOCATE(temp(1:SIZE(work)+lsub-lt))
        ! copy work into temp replacing this occurrence of target by
        ! substring
        temp(1:ipow-1) = work(1:ipow-1)
        temp(ipos:ipos+lsub-1) = substring%chars
        temp(ipos+lsub:ipow) = work(ipos+lt:) 
        work => temp ! make new version of work point at the temp space
        IF(.NOT.rep_search)EXIT ! exit if only first replacement wanted
        ! move search and replacement positions over the effected positions
        ipow = ipos-lt+1
      ENDIF
      ipos=ipos+1
    ENDDO
  ELSE ! forward search
    ipos = 1; ipow = 1
    DO
      IF( ipos > ls-lt+1 ) EXIT ! search past end of string
      IF( ALL(string%chars(ipos:ipos+lt-1) == tget) ) THEN
        ! match found allocate space for string with this occurrence of
        ! target replaced by substring
        ALLOCATE(temp(1:SIZE(work)+lsub-lt))
        ! copy work into temp replacing this occurrence of target by
        ! substring
        temp(1:ipow-1) = work(1:ipow-1)
temp(ipow:ipow+lsub-1) = substring%chars
work(ipow+lsub:) = work(ipow+lt:)

IF(.NOT.rep_search)EXIT ! exit if only first replacement wanted
ipos = ipos+lt-1; ipow = ipow+lsub-1
ENDIF
ipos=ipos+1; ipow=ipow+1
ENDDO
replace_scs%chars => work
ENDFUNCTION replace_scs

FUNCTION replace_scc(string,target,substring,every,back)
type(VARYING_STRING) :: replace_scc
 type(VARYING_STRING), INTENT(IN) :: string
 CHARACTER(LEN=*), INTENT(IN) :: target, substring
 LOGICAL, INTENT(IN), OPTIONAL :: every, back

! calculates the result string by the following actions:
! searches for occurrences of target in string, and replaces these with
! substring. if back present with value true search is backward otherwise
! search is done forward. if every present with value true all accuraces
! of target in string are replaced, otherwise only the first found is
! replaced. if target is not found the result is the same as string.
 LOGICAL :: dir_switch, rep_search
 CHARACTER, DIMENSION(:), POINTER :: work, temp, tget
 INTEGER :: ls, lt, lsub, ipos, ipow
 ls = LEN(string); lt = LEN(target); lsub = LEN(substring)

IF(lt==0)THEN
WRITE(*,*) " Zero length target in REPLACE"
STOP
ENDIF
ALLOCATE(work(1:ls)); work = string%chars
ALLOCATE(tget(1:lt))
DO i=1,lt
  tget(i) = target(i:i)
ENDDO
IF( PRESENT(back) )THEN
  dir_switch = back
ELSE
  dir_switch = .FALSE.
ENDIF
IF( PRESENT(every) )THEN
  rep_search = every
ELSE
  rep_search = .FALSE.
ENDIF
IF( dir_switch )THEN ! backwards search
  ipos = ls-lt+1
  DO 
    IF( ipos < 1 )EXIT ! search past start of string
    ! test for occurance of target in string at this position
    IF( ALL(string%chars(ipos:ipos+lt-1) == tget) )THEN
      ! match found allocate space for string with this occurance of
      ! target replaced by substring
      ALLOCATE(temp(1:SIZE(work)+lsub-lt))
      ! copy work into temp replacing this occurance of target by
      ! substring
      temp(1:i-1) = work(1:i-1)
      DO i=1,lsub
        temp(i+ipos-1) = substring(i:i)
      ENDDO
      temp(ipos+lsub:) = work(ipos+lt:)
      work => temp ! make new version of work point at the temp space
      IF(.NOT.rep_search)EXIT ! exit if only first replacement wanted
      ! move search and replacement positions over the effected positions
      ipos = ipos+lt+1
    ENDIF
  ENDDO
  ipos=ipos+1
ENDIF
ELSE ! forward search
  ipos = 1; ipow = 1
DO
  IF( ipos > ls-lt+1 )EXIT ! search past end of string
  ! test for occurrence of target in string at this position
  IF( ALL(str%chars(ipos:ipos+lt-1) == tget) )THEN
    ! match found allocate space for string with this occurrence of
    ! target replaced by substring
    ALLOCATE(temp(1:SIZE(work)+lsub-lt))
    ! copy work into temp replacing this occurrence of target by
    ! substring
    temp(1:ipow-1) = work(1:ipow-1)
    DO i=1,lsub
      temp(i+ipow-1) = substring(i:i)
    ENDDO
    temp(ipow+lsub:) = work(ipow+lt:)
    ! move search and replacement positions over the effected positions
    ipos = ipos+lt-1; ipow = ipow+lsub-1
  ENDIF
  ipos=ipos+1; ipow=ipow+1
ENDDO
ENDIF
replace_scc%chars => work
ENDFUNCTION replace_scc
FUNCTION replace_css(string,target,substring,every,back)
  type(VARYING_STRING) :: replace_css
  CHARACTER(LEN=*) :: string
  type(VARYING_STRING),INTENT(IN) :: target,substring
  LOGICAL,INTENT(IN),OPTIONAL :: every,back
  ! calculates the result string by the following actions:
  ! searches for occurrences of target in string, and replaces these with
  ! substring. if back present with value true search is backward otherwise
  ! search is done forward. if every present with value true all accurances
  ! of target in string are replaced, otherwise only the first found is
  ! replaced. if target is not found the result is the same as string.
  LOGICAL :: dir_switch, rep_search
  CHARACTER,DIMENSION(:),POINTER :: work,temp,str
  INTEGER :: ls,lt,lsub,ipos,ipow
  ls = LEN(string); lt = LEN(target); lsub = LEN(substring)
  IF(lt==0)THEN
    WRITE(*,*) " Zero length target in REPLACE"
    STOP
  ENDIF
  ALLOCATE(work(1:ls)); ALLOCATE(str(1:ls))
  DO i=1,ls
    str(i) = string(i:i)
  ENDDO
  work = str
  IF( PRESENT(back) )THEN
    dir_switch = back
  ELSE
    dir_switch = .FALSE.
  ENDIF
  IF( PRESENT(every) )THEN
    rep_search = every
  ELSE
    rep_search = .FALSE.
  ENDIF
  IF( dir_switch )THEN ! backwards search
    ipos = ls-lt+1
    DO
      IF( ipos < 1 )EXIT ! search past start of string
      ! test for occurrence of target in string at this position
      IF( ALL(str%chars(ipos:ipos+lt-1) == tget) )THEN
        ! match found allocate space for string with this occurrence of
        ! target replaced by substring
        ALLOCATE(temp(1:SIZE(work)+lsub-lt))
        ! copy work into temp replacing this occurrence of target by
        ! substring
        temp(1:ipow-1) = work(1:ipow-1)
        temp(ipow+lsub:) = work(ipow+lt:)
      ENDIF
      ipos = ipos+lt-1; ipow = ipow+lsub-1
    ENDDO
  ELSE
    ipos=ipos+1; ipow=ipow+1
  ENDIF
  WRITE(*,*) " Zero length target in REPLACE"
STOP
temp(ipos+lsub:) = work(ipos+lt:)
work => temp ! make new version of work point at the temp space
IF(.NOT.rep_search)EXIT ! exit if only first replacement wanted
ipos = ipos+lt+1
ENDIF
ipos=ipos+1
ENDDO
ELSE ! forward search
ipos = 1; ipow = 1
DO
IF( ipos > ls-lt+1 )EXIT ! search past end of string
! test for occurance of target in string at this position
IF( ALL(str(ipos:ipos+lt-1) == target%chars) )THEN
! match found allocate space for string with this occurance of
! target replaced by substring
ALLOCATE(temp(1:SIZE(work)+lsub-lt))
! copy work into temp replacing this occurance of target by
! substring
temp(ipow:ipow+lsub-lt) = substring%chars
temp(ipow+lsub:) = work(ipow+lt:)
work => temp ! make new version of work point at the temp space
ipos = ipos+lt-1; ipow = ipow+lsub-1
ENDIF
ipos=ipos+1; ipow=ipow+1
ENDDO
ENDIF
replace_css%chars => work
ENDFUNCTION replace_css

FUNCTION replace_csc(string,target,substring,every,back)
type(VARYING_STRING) :: replace_csc
CHARACTER(LEN=*),INTENT(IN) :: target
CHARACTER(LEN=*) ,INTENT(IN) :: string,substring
LOGICAL,INTENT(IN),OPTIONAL :: every,back
! calculates the result string by the following actions:
! searches for occurences of target in string, and replaces these with
! substring. if back present with value true search is backward otherwise
! search is done forward. if every present with value true all accurcences
! of target in string are replaced, otherwise only the first found is
! replaced. if target is not found the result is the same as string.
LOGICAL :: dir_switch, rep_search
INTEGER :: ls,lt,lsub,ipos,ipow
ls = LEN(string); lt = LEN(target); lsub = LEN(substring)
IF(lt==0)THEN
WRITE(*,*) " Zero length target in REPLACE"
STOP
ENDIF
ALLOCATE(work(1:ls)); ALLOCATE(str(1:ls))
DO i=1,ls
str(i) = string(i:i)
ENDDO
work = str
IF( PRESENT(back) )THEN
dir_switch = back
ELSE
dir_switch = .FALSE.
ENDIF
IF( PRESENT(every) )THEN
rep_search = every
ELSE
rep_search = .FALSE.
ENDIF
IF( dir_switch )THEN ! backwards search
ipos = ls-lt+1
DO
IF( ipos < 1 )EXIT ! search past start of string
! test for occurance of target in string at this position
IF ALL(str(ipos:ipos+lt-1) == target%chars) THEN
    ! match found allocate space for string with this occurrence of
    ! target replaced by substring
    ALLOCATE(temp(1:SIZE(work)+lsub-lt))
    ! copy work into temp replacing this occurrence of target by
    ! substring
    temp(1:ipos-1) = work(1:ipos-1)
    DO i=1,lsub
      temp(i+ipos-1) = substring(i:i)
    ENDDO
    temp(ipos+lsub:) = work(ipos+lt:)
    work => temp ! make new version of work point at the temp space
    IF(.NOT.rep_search)EXIT ! exit if only first replacement wanted
    ! move search and replacement positions over the effected positions
    ipos = ipos-lt+1
ENDIF
ipos=ipos-1
ENDDO
ELSE ! forward search
  ipos = 1; ipow = 1
  DO
    IF( ipos > ls-lt+1 )EXIT ! search past end of string
    ! test for occurrence of target in string at this position
    IF ALL(str(ipos:ipos+lt-1) == target%chars) THEN
      ! match found allocate space for string with this occurrence of
      ! target replaced by substring
      ALLOCATE(temp(1:SIZE(work)+lsub-lt))
      ! copy work into temp replacing this occurrence of target by
      ! substring
      temp(1:ipow-1) = work(1:ipow-1)
      DO i=1,lsub
        temp(i+ipow-1) = substring(i:i)
      ENDDO
      temp(ipow+lsub:) = work(ipow+lt:)
      work => temp ! make new version of work point at the temp space
      IF(.NOT.rep_search)EXIT ! exit if only first replacement wanted
      ! move search and replacement positions over the effected positions
      ipos = ipos+lt-1; ipow = ipow+lsub-1
    ENDF
    ipos=ipos+1; ipow=ipow+1
  ENDDO
ENDIF
replace_csc%chars => work
ENDFUNCTION replace_csc
FUNCTION replace_ccs(string,target,substring,every,back)
type(VARYING_STRING) :: replace_ccs
  type(VARYING_STRING),INTENT(IN) :: substring
  CHARACTER(LEN=*) ,INTENT(IN) :: string,target
  LOGICAL,INTENT(IN),OPTIONAL :: every,back
  ! calculates the result string by the following actions:
  ! searches for occurrences of target in string, and replaces these with
  ! substring. if back present with value true search is backward otherwise
  ! search is done forward. if every present with value true all accurances
  ! of target in string are replaced, otherwise only the first found is
  ! replaced. if target is not found the result is the same as string.
  LOGICAL :: dir_switch, rep_search
  CHARACTER,DIMENSION(:),POINTER :: work,temp
  INTEGER :: ls,lt,lsub,ipos,ipow
  ls = LEN(string); lt = LEN(target); lsub = LEN(substring)
  IF(lt==0)THEN
    WRITE(*,*) " Zero length target in REPLACE"
    STOP
  ENDIF
  ALLOCATE(work(1:ls))
  DO i=1,ls
    work(i) = string(i:i)
  ENDDO
  IF( PRESENT(back) )THEN
    dir_switch = back
  ELSE
    dir_switch = .FALSE.
  ENDIF
  !
ENDIF
1 IF( PRESENT(every) ) THEN
2 rep_search = every
3 ELSE
4 rep_search = .FALSE.
5 ENDIF
6 IF( dir_switch ) THEN ! backwards search
7 ipos = ls-lt+1
8 DO
9 IF( ipos < 1 ) EXIT ! search past start of string
10 ! test for occurrence of target in string at this position
11 IF( string(ipos:ipos+lt-1) == target ) THEN
12 ! match found allocate space for string with this occurrence of
13 ! target replaced by substring
14 ALLOCATE(temp(1:SIZE(work)+lsub-lt))
15 ! copy work into temp replacing this occurrence of target by
16 ! substring
17 temp(1:ipos-1) = work(1:ipos-1)
18 temp(ipos:ipos+lsub-1) = substring%chars
19 temp(ipos+lsub:) = work(ipos+lt:)
20 work => temp ! make new version of work point at the temp space
21 ! move search and replacement positions over the effected positions
22 ipos = ipos-lt+1
23 ENDIF
24 ipos=ipos-1
25 ENDDO
26 ELSE ! forward search
27 ipos = 1; ipow = 1
28 DO
29 IF( ipos > ls-lt+1 ) EXIT ! search past end of string
30 ! test for occurrence of target in string at this position
31 IF( string(ipos:ipos+lt-1) == target ) THEN
32 ! match found allocate space for string with this occurrence of
33 ! target replaced by substring
34 ALLOCATE(temp(1:SIZE(work)+lsub-lt))
35 ! copy work into temp replacing this occurrence of target by
36 ! substring
37 temp(1:ipow-1) = work(1:ipow-1)
38 temp(ipow:ipow+lsub-1) = substring%chars
39 temp(ipow+lsub:) = work(ipow+lt:)
40 work => temp ! make new version of work point at the temp space
41 ! move search and replacement positions over the effected positions
42 ipos = ipos+lt-1; ipow = ipow+lsub-1
43 ENDIF
44 ipos=ipos+1; ipow=ipow+1
45 ENDDO
46 replace_ccs%chars => work
47 ENDFUNCTION replace_ccs
48 FUNCTION replace_ccc(string, target, substring, every, back)
49 type(VARYING_STRING) :: replace_ccc
50 CHARACTER(LEN=*) , INTENT(IN) :: string, target, substring
51 LOGICAL, INTENT(IN), OPTIONAL :: every, back
52 ! calculates the result string by the following actions:
53 ! searches for occurrences of target in string, and replaces these with
54 ! substring. if back present with value true search is backward otherwise
55 ! search is done forward. if every present with value true all occurrences
56 ! of target in string are replaced, otherwise only the first found is
57 ! replaced. if target is not found the result is the same as string.
58 LOGICAL :: dir_switch, rep_search
59 CHARACTER, DIMENSION(:), POINTER :: work, temp
60 INTEGER :: ls, lt, lsub, ipos, ipow
61 ls = LEN(string); lt = LEN(target); lsub = LEN(substring)
62 IF(lt==0) THEN
63 STOP
64 ENDIF
65 ALLOCATE(work(1:ls))
66 DO i=1,ls
67
work(i) = string(i:i)
ENDO

IF( PRESENT(back) ) THEN
  dir_switch = back
ELSE
  dir_switch = .FALSE.
ENDIF

IF( PRESENT(every) ) THEN
  rep_search = every
ELSE
  rep_search = .FALSE.
ENDIF

IF( dir_switch ) THEN ! backwards search
  ipos = ls-lt+1
  DO
    IF( ipos < 1 ) EXIT ! search past start of string
    IF( string(ipos:ipos+lt-1) == target ) THEN
      ! match found allocate space for string with this occurrence of
      ! target replaced by substring
      ALLOCATE(temp(1:SIZE(work)+lsub-lt))
      ! copy work into temp replacing this occurrence of target by
      ! substring
      temp(1:ipos-1) = work(1:ipos-1)
      DO i=1,lsub
        temp(i+ipos-1) = substring(i:i)
      ENDDO
      temp(ipos+lsub:) = work(ipos+lt:)
      work => temp ! make new version of work point at the temp space
      IF(.NOT.rep_search) EXIT ! exit if only first replacement wanted
      ! move search and replacement positions over the effected positions
      ipos = ipos-lt+1
    ENDDO
    ipos=ipos-1
  ENDDO
ELSE ! forward search
  ipos=1; ipow=1
  DO
    IF( ipos > ls-lt+1 ) EXIT ! search past end of string
    IF( string(ipos:ipos+lt-1) == target ) THEN
      ! match found allocate space for string with this occurrence of
      ! target replaced by substring
      ALLOCATE(temp(1:SIZE(work)+lsub-lt))
      ! copy work into temp replacing this occurrence of target by
      ! substring
      temp(1:ipow-1) = work(1:ipow-1)
      DO i=1,lsub
        temp(i+ipow-1) = substring(i:i)
      ENDDO
      temp(ipow+lsub:) = work(ipow+lt:)
      work => temp ! make new version of work point at the temp space
      IF(.NOT.rep_search) EXIT ! exit if only first replacement wanted
      ! move search and replacement positions over the effected positions
      ipos = ipos+lt-1; ipow = ipow+lsub-1
    ENDDO
    ipos=ipos+1; ipow=ipow+1
  ENDDO
ENDIF
ENDFUNCTION replace_ccc

!----- Remove procedures ----------------------------------------------------!
FUNCTION remove_s(string,start,finish)
  type(VARYING_STRING) :: remove_s
  type(VARYING_STRING), INTENT(IN) :: string
  INTEGER, INTENT(IN), OPTIONAL :: start
  INTEGER, INTENT(IN), OPTIONAL :: finish
  ! returns as result the string produced by the actions
  ! removes the characters between start and finish from string reducing it in
  ! size by MAX(0,ABS(finish-start+1))
  ! if start < 1 or is missing then assumes start=1
! if finish > LEN(string) or is missing then assumes finish=LEN(string)

CHARACTER,DIMENSION(:),POINTER :: arg_str
INTEGER :: is,if,ls
ls = LEN(string)
IF (PRESENT(start)) THEN
   is = MAX(1,start)
ELSE
   is = 1
ENDIF
IF (PRESENT(finish)) THEN
   if = MIN(ls,finish)
ELSE
   if = ls
ENDIF
IF( if < is ) THEN ! zero characters to be removed, string is unchanged
   ALLOCATE(arg_str(1:ls))
   arg_str = string%chars
ELSE
   ALLOCATE(arg_str(1:ls-if+is-1) )
   arg_str(1:is-1) = string%chars(1:is-1)
   arg_str(is:) = string%chars(if+1:)
ENDIF
remove_s%chars => arg_str
ENDFUNCTION remove_s

FUNCTION remove_c(string,start,finish)
type(VARYING_STRING) :: remove_c
CHARACTER(LEN=*) :: string
INTEGER,INTENT(IN),OPTIONAL :: start
INTEGER,INTENT(IN),OPTIONAL :: finish
!
! returns as result the string produced by the actions
! removes the characters between start and finish from string reducing it in
! size by MAX(0,ABS((finish-start+1))
! if start < 1 or is missing then assumes start=1
! if finish > LEN(string) or is missing then assumes finish=LEN(string)
CHARACTER,DIMENSION(:),POINTER :: arg_str
INTEGER :: is,if,ls
ls = LEN(string)
IF (PRESENT(start)) THEN
   is = MAX(1,start)
ELSE
   is = 1
ENDIF
IF (PRESENT(finish)) THEN
   if = MIN(ls,finish)
ELSE
   if = ls
ENDIF
IF( if < is ) THEN ! zero characters to be removed, string is unchanged
   ALLOCATE(arg_str(1:ls))
   DO i=1,ls
      arg_str(i) = string(i:i)
   ENDDO
ELSE
   ALLOCATE(arg_str(1:ls-if+is-1) )
   DO i=1,is-1
      arg_str(i) = string(i:i)
   ENDDO
   DO i=is,ls-if+is-1
      arg_str(i) = string(i-is+if+1:i-is+if+1)
   ENDDO
ENDIF
remove_c%chars => arg_str
ENDFUNCTION remove_c
! delivers these as the result of the function, string is unchanged
! if start < 1 or is missing then it is treated as 1
! if finish > LEN(string) or is missing then it is treated as LEN(string)
INTEGER :: is, if
IF (PRESENT(start)) THEN
  is = MAX(1, start)
ELSE
  is = 1
ENDIF
IF (PRESENT(finish)) THEN
  if = MIN(LEN(string), finish)
ELSE
  if = LEN(string)
ENDIF
ALLOCATE(extract_s%chars(1:if-is+1))
extract_s%chars = string%chars(is:if)
ENDFUNCTION extract_s

FUNCTION extract_c(string, start, finish)
  CHARACTER(LEN=*) , INTENT(IN) :: string
  INTEGER, INTENT(IN), OPTIONAL :: start
  INTEGER, INTENT(IN), OPTIONAL :: finish
  type(VARYING_STRING) :: extract_c
  ! extracts the characters between start and finish from character string and
  ! delivers these as the result of the function, string is unchanged
  ! if start < 1 or is missing then it is treated as 1
  ! if finish > LEN(string) or is missing then it is treated as LEN(string)
  INTEGER :: is, if
  IF (PRESENT(start)) THEN
    is = MAX(1, start)
  ELSE
    is = 1
  ENDIF
  IF (PRESENT(finish)) THEN
    if = MIN(LEN(string), finish)
  ELSE
    if = LEN(string)
  ENDIF
  ALLOCATE(extract_c%chars(1:if-is+1))
  DO i=is, if
    extract_c%chars(i-is+1) = string(i:i)
  ENDDO
ENDFUNCTION extract_c

!----- Split procedures ------------------------------------------------------!
SUBROUTINE split_s(string, word, set, separator, back)
  type(VARYING_STRING), INTENT(INOUT) :: string
  type(VARYING_STRING), INTENT(OUT) :: word
  type(VARYING_STRING), INTENT(IN) :: set
  type(VARYING_STRING), INTENT(OUT), OPTIONAL :: separator
  LOGICAL, INTENT(IN), OPTIONAL :: back
  ! splits the input string at the first(last) character in set
  ! returns the leading(trailing) substring in word and the trailing(leading)
  ! substring in string. The search is done in the forward or backward
  ! direction depending on back. If separator is present, the actual separator
  ! character found is returned in separator.
  ! If no character in set is found string and separator are returned as
  ! zero length and the whole input string is returned in word.
  LOGICAL :: dir_switch
  INTEGER :: ls, tpos
  ls = LEN(string)
  IF( PRESENT(back) ) THEN
    dir_switch = back
  ELSE
    dir_switch = .FALSE.
  ENDIF
  IF(dir_switch) THEN ! backwards search
    DO tpos = ls, 1, -1
      IF(ANY(string%chars(tpos) == set%chars)) EXIT
    ENDDO
    word%chars => string%chars(tpos+1:ls)
    IF(PRESENT(separator)) THEN
      ALLOCATE(word%chars(1:ls))
      word%chars(1:ls) = string%chars(tpos+1:ls)
    ELSE
IF (tpos == 0) THEN
  separator = ""
ELSE
  separator%chars => string%chars(tpos:tpos)
ENDIF
ENDIF
string%chars => string%chars(1:tpos-1)
ELSE ! forwards search
  DO tpos = 1, ls
    IF (ANY (string%chars(tpos) == set%chars)) EXIT
  ENDDO
  word%chars => string%chars(1:tpos-1)
  IF (PRESENT (separator)) THEN
    IF (tpos == ls+1) THEN
      separator = ""
    ELSE
      separator%chars => string%chars(tpos:tpos)
    ENDIF
    ENDIF
  string%chars => string%chars(tpos+1:ls)
ENDIF
ENDIF
ENDSUBROUTINE split_s

SUBROUTINE split_c(string, word, set, separator, back)
type (VARYING_STRING), INTENT (INOUT) :: string
type (VARYING_STRING), INTENT (OUT) :: word
CHARACTER (LEN=*) , INTENT (IN) :: set
! splits the input string at the first (last) character in set
! returns the leading (trailing) substring in word and the trailing (leading)
! substring in string. The search is done in the forward or backward
! direction depending on back. If separator is present, the actual separator
! character found is returned in separator.
! If no character in set is found string and separator are returned as
! zero length and the whole input string is returned in word.
LOGICAL :: dir_switch
INTEGER :: ls, tpos, lset
ls = LEN(string); lset = LEN(set)
IF (PRESENT (back)) THEN
  dir_switch = back
ELSE
  dir_switch = .FALSE.
ENDIF
IF (dir_switch) THEN ! backwards search
  BSEARCH: DO tpos = ls, 1, -1
    DO i = 1, lset
      IF (string%chars(tpos) == set(i:i)) EXIT BSEARCH
    ENDDO
  ENDDO BSEARCH
  word%chars => string%chars(tpos+1:ls)
  IF (PRESENT (separator)) THEN
    IF (tpos == 0) THEN
      separator = ""
    ELSE
      separator%chars => string%chars(tpos:tpos)
    ENDIF
  ELSE
    separator%chars => string%chars(tpos:tpos)
  ENDIF
ENDIF
ELSE ! forwards search
  FSEARCH: DO tpos = 1, ls
    DO i = 1, lset
      IF (string%chars(tpos) == set(i:i)) EXIT FSEARCH
    ENDDO
  ENDDO FSEARCH
  word%chars => string%chars(1:tpos-1)
  IF (PRESENT (separator)) THEN
    IF (tpos == ls+1) THEN
      separator = ""
    ELSE
      separator%chars => string%chars(tpos:tpos)
    ENDIF
  ELSE
    separator%chars => string%chars(tpos:tpos)
  ENDIF
ENDIF
ENDIF
  string%chars => string%chars(tpos+1:ls)
ENDIF
ENDSUBROUTINE split_c

!*----- INDEX procedures -----------------------------------------------!*
FUNCTION index_ss(string,substring,back)
type(VARYING_STRING),INTENT(IN) :: string,substring
LOGICAL,INTENT(IN),OPTIONAL :: back
INTEGER :: index_ss
! returns the starting position in string of the substring
! scanning from the front or back depending on the logical argument back
LOGICAL :: dir_switch
INTEGER :: ls,lsub
ls = LEN(string); lsub = LEN(substring)
IF( PRESENT(back) )THEN
dir_switch = back
ELSE
dir_switch = .FALSE.
ENDIF
IF(dir_switch)THEN ! backwards search
  DO i = ls-lsub+1,1,-1
    IF( ALL(string%chars(i:i+lsub-1) == substring%chars) )THEN
      index_ss = i
      RETURN
    ENDIF
  ENDDO
  index_ss = 0
ELSE ! forward search
  DO i = 1,ls-lsub+1
    IF( ALL(string%chars(i:i+lsub-1) == substring%chars) )THEN
      index_ss = i
      RETURN
    ENDIF
  ENDDO
  index_ss = 0
ENDIF
ENDFUNCTION index_ss

FUNCTION index_sc(string,substring,back)
type(VARYING_STRING),INTENT(IN) :: string
CHARACTER(LEN=*) ,INTENT(IN) :: substring
LOGICAL,INTENT(IN),OPTIONAL :: back
INTEGER :: index_sc
! returns the starting position in string of the substring
! scanning from the front or back depending on the logical argument back
LOGICAL :: dir_switch,matched
INTEGER :: ls,lsub
ls = LEN(string); lsub = LEN(substring)
IF( PRESENT(back) )THEN
dir_switch = back
ELSE
dir_switch = .FALSE.
ENDIF
IF (dir_switch) THEN ! backwards search
  DO i = ls-lsub+1,1,-1
    matched = .TRUE.
    DO j = 1,lsub
      IF( string%chars(i+j-1) /= substring(j:j) )THEN
        matched = .FALSE.
        EXIT
      ENDIF
    ENDDO
    IF( matched )THEN
      index_sc = i
      RETURN
    ENDIF
  ENDDO
  index_sc = 0
ELSE ! forward search
  DO i = 1,ls-lsub+1
    matched = .TRUE.
    DO j = 1,lsub
      IF( string%chars(i+j-1) /= substring(j:j) )THEN
        matched = .FALSE.
        EXIT
      ENDIF
    ENDDO
  IF( matched )THEN
    index_sc = i
    RETURN
  ENDIF
  ENDDO
  index_sc = 0
ENDIF
ENDFUNCTION index_sc
DO j = 1,lsub
   IF( string%chars(i+j-1) /= substring(j:j) )THEN
      matched = .FALSE.
      EXIT
   ENDIF
ENDDO
IF( matched )THEN
   index_sc = i
   RETURN
ENDIF
ENDDO
index_sc = 0
ENDIF
ENDFUNCTION index_sc

FUNCTION index_cs(string,substring,back)
   CHARACTER(LEN=*) ,INTENT(IN) :: string
   type(VARYING_STRING),INTENT(IN) :: substring
   LOGICAL,INTENT(IN),OPTIONAL :: back
   INTEGER :: index_cs
   ! returns the starting position in string of the substring
   ! scanning from the front or back depending on the logical argument back
   LOGICAL :: dir_switch,matched
   INTEGER :: ls,lsub
   ls = LEN(string); lsub = LEN(substring)
   IF( PRESENT(back) )THEN
      dir_switch = back
   ELSE
      dir_switch = .FALSE.
   ENDIF
   IF(dir_switch)THEN ! backwards search
      DO i = ls-lsub+1,1,-1
         matched = .TRUE.
         DO j = 1,lsub
            IF( string(i+j-1:i+j-1) /= substring%chars(j) )THEN
               matched = .FALSE.
               EXIT
            ENDIF
         ENDDO
         IF( matched )THEN
            index_cs = i
            RETURN
         ENDIF
      ENDDO
      index_cs = 0
   ELSE ! forward search
      DO i = 1,ls-lsub+1
         matched = .TRUE.
         DO j = 1,lsub
            IF( string(i+j-1:i+j-1) /= substring%chars(j) )THEN
               matched = .FALSE.
               EXIT
            ENDIF
         ENDDO
         IF( matched )THEN
            index_cs = i
            RETURN
         ENDIF
      ENDDO
      index_cs = 0
   ENDIF
ENDFUNCTION index_cs

!----- SCAN procedures ------------------------------------------------------!
FUNCTION scan_ss(string,set,back)
   type(VARYING_STRING),INTENT(IN) :: string,set
   LOGICAL,INTENT(IN),OPTIONAL :: back
   INTEGER :: scan_ss
   ! returns the first position in string occupied by a character from
   ! the characters in set, scanning is forward or backwards depending on back
   LOGICAL :: dir_switch
   INTEGER :: ls
   !----- SCAN procedures ------------------------------------------------------!
FUNCTION scan_ss(string,set,back)
   type(VARYING_STRING),INTENT(IN) :: string,set
   LOGICAL,INTENT(IN),OPTIONAL :: back
   INTEGER :: scan_ss
   ! returns the first position in string occupied by a character from
   ! the characters in set, scanning is forward or backwards depending on back
   LOGICAL :: dir_switch
   INTEGER :: ls
59
CHARACTER::tmp ! inserted to work round a temporary bug in F90 1.1
ls = LEN(string)
IF(PRESENT(back))THEN
  dir_switch = back
ELSE
  dir_switch = .FALSE.
ENDIF
IF(dir_switch)THEN ! backwards search
  DO i = ls,1,-1
    tmp = string%chars(i) ! bug work round
    IF(ANY(set%chars == tmp))THEN
      scan_ss = i
      RETURN
    ENDFIND
ENDDO
ELSE ! forward search
  scan_ss = 0
  DO i = 1,ls
    tmp = string%chars(i) ! bug work round
    IF(ANY(set%chars == tmp))THEN
      scan_ss = i
      RETURN
    ENDFIND
ENDDO
scan_ss = 0
ENDIF
ENDFUNCTION scan_ss

FUNCTION scan_sc(string,set,back)
type(VARYING_STRING),INTENT(IN) :: string
CHARACTER(LEN=*) ,INTENT(IN) :: set
LOGICAL,INTENT(IN),OPTIONAL :: back
INTEGER :: scan_sc
! returns the first position in string occupied by a character from set
! the characters in set, scanning is forward or backwards depending on back
LOGICAL :: dir_switch,matched
INTEGER :: ls
ls = LEN(string)
IF(PRESENT(back))THEN
dir_switch = back
ELSE
  dir_switch = .FALSE.
ENDIF
IF(dir_switch)THEN ! backwards search
  DO i = ls,1,-1
    matched = .FALSE.
    DO j = 1,LEN(set)
      IF(string%chars(i) == set(j:j))THEN
        matched = .TRUE.
        EXIT
      ENDFIND
    ENDDO
    IF(matched)THEN
      scan_sc = i
      RETURN
    ENDFIND
  ENDDO
  scan_sc = 0
ELSE ! forward search
  Do i = 1,ls
    matched = .FALSE.
    DO j = 1,LEN(set)
      IF(string%chars(i) == set(j:j))THEN
        matched = .TRUE.
        EXIT
      ENDFIND
    ENDDO
    IF(matched)THEN
      scan_sc = i
      RETURN
    ENDFIND
  ENDDO
END IF
ENDFUNCTION scan_sc
FUNCTION scan_sc(string, set, back)
  CHARACTER(LEN=*) :: string
  type (VARYING_STRING) :: set
  LOGICAL, INTENT(IN), OPTIONAL :: back
  INTEGER :: scan_sc
ENDFUNCTION scan_sc

FUNCTION scan_cs(string, set, back)
  CHARACTER(LEN=*) :: string
  type (VARYING_STRING) :: set
  LOGICAL, INTENT(IN), OPTIONAL :: back
  INTEGER :: scan_cs
ENDFUNCTION scan_cs

! returns the first position in character string occupied by a character from
! the characters in set, scanning is forward or backwards depending on back
LOGICAL :: dir_switch, matched
INTEGER :: ls
ls = LEN(string)
IF( PRESENT(back) )THEN
  dir_switch = back
ELSE
  dir_switch = .FALSE.
ENDIF
IF(dir_switch)THEN ! backwards search
  DO i = ls,1,-1
    matched = .FALSE.
    DO j = 1,LEN(set)
      IF( string(i:i) == set%chars(j) )THEN
        matched = .TRUE.
        EXIT
      ENDDO
    IF( matched )THEN
      scan_cs = i
      RETURN
    ENDDO
  ENDIF
  scan_cs = 0
ELSE ! forward search
  DO i = 1,ls
    matched = .FALSE.
    DO j = 1,LEN(set)
      IF( string(i:i) == set%chars(j) )THEN
        matched = .TRUE.
        EXIT
      ENDDO
    IF( matched )THEN
      scan_cs = i
      RETURN
    ENDIF
  ENDDO
  scan_cs = 0
ENDIF
ENDFUNCTION scan_cs

!----- VERIFY procedures ----------------------------------------------------!
FUNCTION verify_ss(string, set, back)
  type (VARYING_STRING) :: string, set
  LOGICAL, INTENT(IN), OPTIONAL :: back
  INTEGER :: verify_ss
ENDFUNCTION verify_ss

! returns the first position in string not occupied by a character from
! the characters in set, scanning is forward or backwards depending on back
LOGICAL :: dir_switch
INTEGER :: ls
ls = LEN(string)
IF( PRESENT(back) )THEN
  dir_switch = back
ELSE
  dir_switch = .FALSE.
ENDIF
IF(dir_switch)THEN ! backwards search
  DO i = ls,1,-1
    tmp=string%chars(i) ! bug work round
    IF( .NOT. (ANY( set%chars == tmp )) )THEN
      matched = .FALSE.
      EXIT
    IF( matched )THEN
      verify_ss = i
      RETURN
    ENDIF
  ENDIF
IF(dir_switch)THEN ! backwards search
  DO i = 1,ls
    matched = .FALSE.
    DO j = 1,LEN(set)
      IF( string(i:i) == set%chars(j) )THEN
        matched = .TRUE.
        EXIT
      ENDDO
    IF( matched )THEN
      verify_ss = i
      RETURN
    ENDIF
  ENDIF
ENDIF
IF(dir_switch)THEN ! backwards search
  DO i = ls,1,-1
    tmp=string%chars(i) ! bug work round
    IF( .NOT. (ANY( set%chars == tmp )) )THEN
      matched = .FALSE.
      EXIT
    IF( matched )THEN
      verify_ss = i
      RETURN
    ENDIF
  ENDIF
ENDIF
IF(dir_switch)THEN ! backwards search
  DO i = 1,ls
    matched = .FALSE.
    DO j = 1,LEN(set)
      IF( string(i:i) == set%chars(j) )THEN
        matched = .TRUE.
        EXIT
      ENDDO
    IF( matched )THEN
      verify_ss = i
      RETURN
    ENDIF
  ENDIF
ENDIF
IF(dir_switch)THEN ! backwards search
  DO i = ls,1,-1
    tmp=string%chars(i) ! bug work round
    IF( .NOT. (ANY( set%chars == tmp )) )THEN
      matched = .FALSE.
      EXIT
    IF( matched )THEN
      verify_ss = i
      RETURN
    ENDIF
  ENDIF
ENDIF
IF(dir_switch)THEN ! backwards search
  DO i = 1,ls
    matched = .FALSE.
    DO j = 1,LEN(set)
      IF( string(i:i) == set%chars(j) )THEN
        matched = .TRUE.
        EXIT
      ENDDO
    IF( matched )THEN
      verify_ss = i
      RETURN
    ENDIF
  ENDIF
```fortran
verify_ss = i
RETURN
ENDIF
ENDDO
ELSE ! forward search
DO i = 1,ls
tmp=string%chars(i) ! bug work round
IF( .NOT.(ANY( set%chars == tmp )) )THEN
  verify_ss = i
  RETURN
ENDIF
ENDDO
ENDIF
ENDIF
ENDFUNCTION verify_ss

FUNCTION verify_sc(string,set,back)
  type(VARYING_STRING),INTENT(IN) :: string
  CHARACTER(LEN=*) ,INTENT(IN) :: set
  LOGICAL,INTENT(IN),OPTIONAL :: back
  INTEGER :: verify_sc
! returns the first position in string not occupied by a character from
! the characters in set, scanning is forward or backwards depending on back
  LOGICAL :: dir_switch
  INTEGER :: ls
  ls = LEN(string)
  IF( PRESENT(back) )THEN
    dir_switch = back
  ELSE
    dir_switch = .FALSE.
  ENDIF
  IF(dir_switch)THEN ! backwards search
    back_string_search:DO i = ls,1,-1
      DO j = 1,LEN(set)
        IF( string%chars(i) == set(j:j) )CYCLE back_string_search
      ENDDO
      verify_sc = i
      RETURN
    ENDDO back_string_search
    ! each string character found in set
    verify_sc = 0
  ELSE ! forward search
    frwd_string_search:DO i = 1,ls
      DO j = 1,LEN(set)
        IF( string%chars(i) == set(j:j) )CYCLE frwd_string_search
      ENDDO
      verify_sc = i
      RETURN
    ENDDO frwd_string_search
    ! each string character found in set
    verify_sc = 0
  ENDIF
ENDIF
ENDFUNCTION verify_sc

FUNCTION verify_cs(string,set,back)
  CHARACTER(LEN=*) ,INTENT(IN) :: string
  type(VARYING_STRING),INTENT(IN) :: set
  LOGICAL,INTENT(IN),OPTIONAL :: back
  INTEGER :: verify_cs
! returns the first position in character string not occupied by a character
! from the characters in set, scanning is forward or backwards depending on back
  LOGICAL :: dir_switch
  INTEGER :: ls
  ls = LEN(string)
  IF( PRESENT(back) )THEN
    dir_switch = back
  ELSE
    dir_switch = .FALSE.
  ENDIF
  IF(dir_switch)THEN ! backwards search
    back_string_search:DO i = ls,1,-1
      DO j = 1,LEN(set)
        IF( string%chars(i) == set(j:j) )CYCLE back_string_search
      ENDDO
      verify_cs = i
      RETURN
    ENDDO back_string_search
    ! each character string character found in set
    verify_cs = 0
  ELSE ! forward search
    frwd_string_search:DO i = 1,ls
      DO j = 1,LEN(set)
        IF( string%chars(i) == set(j:j) )CYCLE frwd_string_search
      ENDDO
      verify_cs = i
      RETURN
    ENDDO frwd_string_search
    ! each character string character found in set
    verify_cs = 0
  ENDIF
ENDIF
ENDFUNCTION verify Cs
```
IF(dir_switch) THEN ! backwards search
    back_string_search: DO i = ls,1,-1
        DO j = 1,LEN(set)
            IF( string(i:i) == set%chars(j) ) CYCLE back_string_search
        ENDDO
        verify_cs = i
        RETURN
    ENDDO back_string_search
    verify_cs = 0
ELSE ! forward search
    frwd_string_search: DO i = 1,ls
        DO j = 1,LEN(set)
            IF( string(i:i) == set%chars(j) ) CYCLE frwd_string_search
        ENDDO
        verify_cs = i
        RETURN
    ENDDO frwd_string_search
    verify_cs = 0
ENDIF
ENDFUNCTION verify_cs

!----- LEN_TRIM procedure -----------------------------------------------------!
FUNCTION len_trim_s(string)
  type(VARYING_STRING), INTENT(IN) :: string
  INTEGER :: len_trim_s
  ! Returns the length of the string without counting trailing blanks
  INTEGER :: ls
  ls=LEN(string)
  len_trim_s = 0
  DO i = ls,1,-1
    IF (string%chars(i) /= BLANK) THEN
      len_trim_s = i
      EXIT
    ENDIF
  ENDDO
ENDFUNCTION len_trim_s

!----- TRIM procedure ---------------------------------------------------------!
FUNCTION trim_s(string)
  type(VARYING_STRING), INTENT(IN) :: string
  type(VARYING_STRING) :: trim_s
  ! Returns the argument string with trailing blanks removed
  INTEGER :: ls,pos
  ls=LEN(string)
  pos=0
  DO i = ls,1,-1
    IF (string%chars(i) /= BLANK) THEN
      pos=i
      EXIT
    ENDIF
  ENDDO
  ALLOCATE(trim_s%chars(1:pos))
  trim_s%chars(1:pos) = string%chars(1:pos)
ENDFUNCTION trim_s

!----- IACHAR interface -------------------------------------------------------!
FUNCTION iachar_s(string)
  type(VARYING_STRING), INTENT(IN) :: string
  INTEGER :: iachar_s
  ! returns the position of the character string in the ISO 646
  ! collating sequence.
  ! string must be of length one
  IF (LEN(string) /= 1) THEN
    WRITE(*,*) " ERROR, argument in IACHAR not of length one"
    STOP
  ENDIF
  iachar_s = IACHAR(string%chars(1))
ENDFUNCTION iachar_s

!----- ICHAR procedure --------------------------------------------------------!
FUNCTION ichar_s(string)
  type(VARYING_STRING), INTENT(IN) :: string

INTEGER :: ichar_s
!
returns the position of character from string in the processor collating
!
sequence.
!
string must be of length one
IF (LEN(string) /= 1) THEN
  WRITE(*,*) " Argument string in ICHAR has to be of length one"
  STOP
ENDIF
ichar_s = ICHAR(string%chars(1))
ENDFUNCTION ichar_s

!----- ADJUSTL procedure -----------------------------------------------!
FUNCTION adjustl_s(string)
type(VARYING_STRING),INTENT(IN) :: string

  type(VARYING_STRING) :: adjustl_s

  ! Returns the string adjusted to the left, removing leading blanks and
  ! inserting trailing blanks
  INTEGER :: ls,pos
  ls=LEN(string)
  DO pos = 1,ls
    IF(string%chars(pos) /= blank) EXIT
  ENDDO
  ! pos now holds the position of the first non-blank character
  ! or ls+1 if all characters are blank
  ALLOCATE(adjustl_s%chars(1:ls))
  adjustl_s%chars(1:ls-pos+1) = string%chars(pos:ls)
  adjustl_s%chars(ls-pos+2:ls) = blank
ENDFUNCTION adjustl_s

!----- ADJUSTR procedure -----------------------------------------------!
FUNCTION adjustr_s(string)
type(VARYING_STRING),INTENT(IN) :: string

  type(VARYING_STRING) :: adjustr_s

  ! Returns the string adjusted to the right, removing trailing blanks
  ! and inserting leading blanks
  INTEGER :: ls,pos
  ls=LEN(string)
  DO pos = ls,1,-1
    IF(string%chars(pos) /= blank) EXIT
  ENDDO
  ! pos now holds the position of the last non-blank character
  ! or zero if all characters are blank
  ALLOCATE(adjustr_s%chars(1:ls))
  adjustr_s%chars(ls-pos+1:ls) = string%chars(1:pos)
  adjustr_s%chars(1:ls-pos) = blank
ENDFUNCTION adjustr_s

ENDMODULE ISO_VARYING_STRING
Annex B

(Informative)

This annex includes some examples illustrating the use of facilities conformant with this International Standard. It should be noted that while every care has been taken by the technical working group to ensure that these example programs are a correct implementation of the stated problems using this International Standard and in valid Fortran code, no guarantee is given or implied that this code will produce correct results, or even that it will execute on any particular processor.

```fortran
PROGRAM word_count
!-----------------------------------------------------------------------------!
! Counts the number of "words" contained in a file. The words are assumed to !
! be terminated by any one of: !
! space,comma,period,!,? , or the EoR !
! The file may have records of any length and the file may contain any number !
! of records. !
! The program prompts for the name of the file to be subject to a word count !
! and the result is written to the default output unit !
!-----------------------------------------------------------------------------!
USE ISO_VARYING_STRING

type(VARYING_STRING) :: line,fname
INTEGER :: ierr,nd,wcount=0
WRITE(*,ADVANCE='NO',FMT='(A)') " Input name of file?"
CALL GET(STRING=fname) ! read the required filename from the default
OPEN(UNIT=1,FILE=CHAR(fname)) ! CHAR(fname) converts to the type
file_read: DO ! until EoF reached
   CALL GET(1,line,IOSTAT=ierr) ! read next line of file
   IF(ierr == -1)EXIT file_read
   word_scan: DO ! until end of line
      nd=SCAN(line," ,.!?") ! scan to find end of word
      IF(nd == 0)THEN ! EoR is end of word
         nd = LEN(line)
      ENDIF
      IF(nd > 0) wcount=wcount+1 ! at least one non-terminator character
         line = REMOVE(line,1,nd) ! strips the counted word and its terminator
    ENDDO word_scan
   IF(nd > 0) wcount=wcount+1
   ENDDO file_read
   IF(ierr < 0)THEN
      WRITE(*,*) "Error in GET file in word_count, No. ",ierr
   ENDIF
ENDPROGRAM word_count

Note, it is not claimed that the above program is the best way to code this problem, nor even that it is a good way, merely that it is a way of solving this simple problem using the facilities defined by this International Standard.

A second and rather more realistic example is one which extends the above trivial example by producing a full vocabulary list along with frequency of occurrence for each different word.

```
The file may have records of any length and the file may contain any number of records.

The program prompts for the name of the file to be subject to a word count and the result is written to the default output unit.

Also builds a list of the vocabulary found and the frequency of occurrence of each different word.

---

```
USE ISO_VARYING_STRING

! -----------------------------------------------!
! Vocabulary list and frequency count arrays. The size of these arrays will be extended dynamically in steps of 100 as the used vocabulary grows. !
! -----------------------------------------------!

! Vocabulary list and frequency count arrays. The size of these arrays will be extended dynamically in steps of 100 as the used vocabulary grows. !

USE ISO_VARYING_STRING

! -----------------------------------------------!
! Vocabulary list and frequency count arrays. The size of these arrays will be extended dynamically in steps of 100 as the used vocabulary grows. !
! -----------------------------------------------!

USE ISO_VARYING_STRING

! -----------------------------------------------!
! Vocabulary list and frequency count arrays. The size of these arrays will be extended dynamically in steps of 100 as the used vocabulary grows. !
! -----------------------------------------------!
```

---

```
USE ISO_VARYING_STRING

! -----------------------------------------------!
! Vocabulary list and frequency count arrays. The size of these arrays will be extended dynamically in steps of 100 as the used vocabulary grows. !
! -----------------------------------------------!
```

---

```
USE ISO_VARYING_STRING

! -----------------------------------------------!
! Vocabulary list and frequency count arrays. The size of these arrays will be extended dynamically in steps of 100 as the used vocabulary grows. !
! -----------------------------------------------!
```

---

```
USE ISO_VARYING_STRING

! -----------------------------------------------!
! Vocabulary list and frequency count arrays. The size of these arrays will be extended dynamically in steps of 100 as the used vocabulary grows. !
! -----------------------------------------------!
```

---

```
USE ISO_VARYING_STRING

! -----------------------------------------------!
! Vocabulary list and frequency count arrays. The size of these arrays will be extended dynamically in steps of 100 as the used vocabulary grows. !
! -----------------------------------------------!
```

---

```
USE ISO_VARYING_STRING

! -----------------------------------------------!
! Vocabulary list and frequency count arrays. The size of these arrays will be extended dynamically in steps of 100 as the used vocabulary grows. !
! -----------------------------------------------!
```

---

```
USE ISO_VARYING_STRING

! -----------------------------------------------!
! Vocabulary list and frequency count arrays. The size of these arrays will be extended dynamically in steps of 100 as the used vocabulary grows. !
! -----------------------------------------------!
```
! and frequency information in the new extended lists

!-----------------------------------------------------------------------------!
type(VARYING_STRING),DIMENSION(list_size) :: vocab_swap
INTEGER,DIMENSION(list_size) :: freq_swap
INTEGER,PARAMETER :: list_increment=100
INTEGER :: new_list_size,alerr
vocab_swap = vocab ! copy old list into temporary space
freq_swap = freq
new_list_size = list_size + list_increment
DEALLOCATE(vocab,freq)
ALLOCATE(vocab(1:new_list_size),freq(1:new_list_size),STAT=alerr)
IF(alerr /= 0)THEN
  WRITE(*,*) "Unable to extend vocabulary list"
  STOP
ENDIF
vocab(1:list_size) = vocab_swap ! copy old list back into bottom
freq(1:list_size) = freq_swap ! of new extended list
list_size = new_list_size
ENDSUBROUTINE extend_lists

SUBROUTINE update_vocab_lists
!-----------------------------------------------------------------------------!
! Accesses the host variables:
! type(VARYING_STRING),ALLOCATABLE,DIMENSION(:) :: vocab
! INTEGER,ALLOCATABLE,DIMENSION(:) :: freq
! INTEGER :: list_size,list_top
! type(VARYING_STRING) :: word
! searches the existing words in vocab to find a match for word
! if found increments the freq if not found adds word to
! list_top + 1 vocab list and sets corresponding freq to 1
! if list_size exceeded extend the list size before updating
!-----------------------------------------------------------------------------!
list_search: DO i=1,list_top
  IF(word == vocab(i))THEN
    freq(i) = freq(i) + 1
    RETURN
  ENDIF
ENDDO list_search
IF(list_top == list_size)THEN
  CALL extend_lists
ENDIF
list_top = list_top + 1
vocab(list_top) = word
freq(list_top) = 1
ENDSUBROUTINE update_vocab_lists

ENDPROGRAM vocabulary_word_count