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

[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 straight forward 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

ISO_VARYING_STRING

Programs shall be able to access the facilities defined by this International Standard by the inclusion of **USE** statements of the form

USE ISO_VARYING_STRING

2.2 The Type

The type shall have the name

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 **PRIVATE** to the module.

2.3 Extended Meanings for Intrinsic Operators

The meanings for the intrinsic operators of:

assignment	**=**
concatenation	**//**
comparisons	**==, /=, <, <=, >=, >**

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

2.3.1 Assignment: An assignment of the form

var = expr

shall be defined for scalars with the following type combinations:

VARYING_STRING = VARYING_STRING
VARYING_STRING = CHARACTER
CHARACTER = VARYING_STRING

Action: The characters that are the value of the expression **expr** become the value of the variable **var**. There are two cases:

Case(i) : Where the variable is of type **VARYING_STRING**, the length of the variable becomes that of the expression.

Case(ii) : Where the variable is of type **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.

2.3.2 Concatenation: The concatenation operation

```
string_a // string_b
```

shall be defined for scalars with the following type combinations:

```
VARYING_STRING // VARYING_STRING
VARYING_STRING // CHARACTER
     CHARACTER // VARYING_STRING
```

The values of the operands are unchanged by the operation.

Result Type: `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

```
string_a .OP. string_b
```

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

```
VARYING_STRING .OP. VARYING_STRING
VARYING_STRING .OP. CHARACTER
     CHARACTER .OP. 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 `LOGICAL`.

Result Value: The result value is true if `string_a` stands in the indicated relation to `string_b`. The collating sequence used for the inequality comparisons is that defined by the processor for characters of default kind. If `string_a` and `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 `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 `VARYING_STRING` and `CHARACTER`.

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

```
LEN(string)
```

shall be added.

Description: Returns the length of a character string.

Class: Transformational function.

Argument: `string` is a scalar of type `VARYING_STRING`. The argument is unchanged by the procedure.

Result Type: default `INTEGER`.

Result Value: The result value is the number of characters in `string`.

2.4.2 The CHAR Procedure: The generic function references of the form

```
CHAR(string)
CHAR(string,length)
```

shall be added.

Description: Converts a varying string value to default character.

Class: Transformational function.

Arguments:

`string` - is of type `VARYING_STRING`

`length` - is of type default `INTEGER`.

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

Result Type: default `CHARACTER`.

Result Value:

Case(i) : If `length` is absent, the result has the value of the characters of `string`, and the same length.

Case(ii) : If `length` is present, the result has the length specified by the argument `length`. If `string` is longer than `length`, the result is truncated on the right. If `string` is shorter than `length`, the result is padded on the right with blanks. If `length` is less than one, the result is of zero length.

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

```
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 `VARYING_STRING` and of length exactly one. The argument is unchanged by the procedure.

Result Type: default `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

```
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 `VARYING_STRING` and of length exactly one. The argument is unchanged by the procedure.

Result Type: default `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

```
TRIM(string)
```

shall be added.

Description: Remove trailing blanks from a string.

Class: Transformational Function.

Argument: `string` is a scalar of type `VARYING_STRING`. The argument is unchanged by the procedure.

Result Type: **`VARYING_STRING`**.
Result Value: The result value is the string produced by removing any trailing blanks from the argument. If the argument **`string`** contains only blank characters or is of zero length, the result is a zero-length string.

2.4.6 The LEN_TRIM procedure: The generic function reference of the form

```
LEN_TRIM(string)
```

shall be added.
Description: Length of a string not counting any trailing blanks.
Class: Transformational function.
Argument: **`string`** is a scalar of type **`VARYING_STRING`**. The argument is unchanged by the procedure.
Result Type: default **`INTEGER`**.
Result Value: The result value is the position of the last non-blank character in **`string`**. If the argument **`string`** contains only blank characters or is of zero length, the result is zero.

2.4.7 The ADJUSTL procedure: The generic function reference of the form

```
ADJUSTL(string)
```

shall be added.
Description: Adjusts to the left, removing any leading blanks and inserting trailing blanks.
Class: Transformational function.
Argument: **`string`** is a scalar of type **`VARYING_STRING`**. The argument is unchanged by the procedure.
Result Type: **`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 **`string`** is non-blank, **`string`** contains only blank characters or is of zero length.

2.4.8 The ADJUSTR procedure: The generic function reference of the form

```
ADJUSTR(string)
```

shall be added.
Description: Adjusts to the right, removing any trailing blanks and inserting leading blanks.
Class: Transformational function.
Argument: **`string`** is a scalar of type **`VARYING_STRING`**. The argument is unchanged by the procedure.
Result Type: **`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 **`string`** is non-blank, **`string`** contains only blank characters or is of zero length.

2.4.9 The REPEAT procedure: The generic function reference of the form

```
REPEAT(string,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

```
Lop(string_a,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

```
INDEX(string,substring,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,
`EXTRACT(string,I,I+LEN(substring)-1)==substring`,
or zero if there is no such value. Zero is returned if `LEN(string)<LEN(substring)`, and one is returned if `LEN(substring)=0`.

Case(ii) : If `back` is present with the value true, the result is the maximum value of `I` less than or equal to `LEN(string)-LEN(substring)+1` such that `EXTRACT(string,I:I+LEN(substring)-1)==substring`,
or zero if there is no such value. Zero is returned if `LEN(string)<LEN(substring)`, and `LEN(string)+1` is returned if `LEN(substring)=0`.

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

```
SCAN(string,set,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

```
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

```
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

```
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

```
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

```
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.

INSERT	-	insert a substring into a string
REPLACE	-	replace a substring in a string
REMOVE	-	remove a section of a string
EXTRACT	-	extract a section from a string
SPLIT	-	split a string into two at the occurrence of a separator

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

`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

`start` to **`MIN(start+LEN(substring)-1,LEN(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

`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

`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

`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 occurence 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 straight forward 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 auxilliary 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.                                                               !
```

```
1       ! Note: this Module is defined only for characters of default kind. A similar !
2       ! module could be defined for non-default characters if these are supported   !
3       ! on a processor by adding a KIND parameter to the component in the type      !
4       ! definition, and to all delarations of objects of CHARACTER type.            !
5       !-------------------------------------------------------------------------------!
6
7       CHARACTER,PARAMETER :: blank = " "
8
9       !----- GENERIC PROCEDURE INTERFACE DEFINITIONS -----------------------------!
10
11      !----- LEN interface -------------------------------------------------------!
12      INTERFACE LEN
13        MODULE PROCEDURE len_s   ! length of string
14      ENDINTERFACE
15
16      !----- Conversion procedure interfaces -------------------------------------!
17      INTERFACE VAR_STR
18        MODULE PROCEDURE c_to_s   ! character to string
19      ENDINTERFACE
20
21      INTERFACE CHAR
22        MODULE PROCEDURE s_to_c, &   ! string to character
23                         s_to_fix_c  ! string to specified length character
24      ENDINTERFACE
25
26      !----- ASSIGNMENT interfaces -----------------------------------------------!
27      INTERFACE ASSIGNMENT(=)
28        MODULE PROCEDURE s_ass_s, &   ! string    = string
29                         c_ass_s, &   ! character = string
30                         s_ass_c      ! string    = character
31      ENDINTERFACE
32
33      !----- Concatenation operator interfaces -----------------------------------!
34      INTERFACE OPERATOR(//)
35        MODULE PROCEDURE s_concat_s, &  ! string//string
36                         s_concat_c, &  ! string//character
37                         c_concat_s     ! character//string
38      ENDINTERFACE
39
40      !----- Repeated Concatenation interfaces -----------------------------------!
41      INTERFACE REPEAT
42        MODULE PROCEDURE repeat_s
43      ENDINTERFACE
44
45      !------ Equality comparison operator interfaces-----------------------------!
46      INTERFACE OPERATOR(==)
47        MODULE PROCEDURE s_eq_s, &  ! string==string
48                         s_eq_c, &  ! string==character
49                         c_eq_s     ! character==string
50      ENDINTERFACE
51
52      !----- not-equality comparison operator interfaces -------------------------!
53      INTERFACE OPERATOR(/=)
54        MODULE PROCEDURE s_ne_s, &  ! string/=string
55                         s_ne_c, &  ! string/=character
56                         c_ne_s     ! character/=string
57      ENDINTERFACE
58
59      !----- less-than comparison operator interfaces ----------------------------!
60      INTERFACE OPERATOR(<)
61        MODULE PROCEDURE s_lt_s, &  ! string<string
62                         s_lt_c, &  ! string<character
63                         c_lt_s     ! character<string
64      ENDINTERFACE
65
66      !----- less-than-or-equal comparison operator interfaces -------------------!
67      INTERFACE OPERATOR(<=)
68        MODULE PROCEDURE s_le_s, &  ! string<=string
69                         s_le_c, &  ! string<=character
70                         c_le_s     ! character<=string
71      ENDINTERFACE
72
```

```
!----- greater-than-or-equal comparison operator interfaces -----------------!
INTERFACE OPERATOR(>=)
  MODULE PROCEDURE s_ge_s, &  ! string>=string
                   s_ge_c, &  ! string>=character
                   c_ge_s     ! character>=string
ENDINTERFACE

!----- greater-than comparison operator interfaces --------------------------!
INTERFACE OPERATOR(>)
  MODULE PROCEDURE s_gt_s, &  ! string>string
                   s_gt_c, &  ! string>character
                   c_gt_s     ! character>string
ENDINTERFACE

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

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

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

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

!----- Input function interface ---------------------------------------------!
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

!----- Output procedure interfaces ------------------------------------------!
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

!----- Insert procedure interfaces ------------------------------------------!
INTERFACE INSERT
  MODULE PROCEDURE insert_ss, & ! string in string
                   insert_sc, & ! char in string
                   insert_cs, & ! string in char
```

```
                    insert_cc    ! char in char
ENDINTERFACE

!----- Replace procedure interfaces ----------------------------------------!
INTERFACE REPLACE
  MODULE PROCEDURE replace_ss, &   ! string by string, at specified
                   replace_sc, &   ! string by char  , starting
                   replace_cs, &   ! char by string  , point
                   replace_cc, &   ! char by char
                   replace_ss_sf,& ! string by string, between
                   replace_sc_sf,& ! string by char  , specified
                   replace_cs_sf,& ! char by string  , starting and
                   replace_cc_sf,& ! char by char    , finishing points
                   replace_sss, &  ! in string replace string by string
                   replace_ssc, &  ! in string replace string by char
                   replace_scs, &  ! in string replace char by string
                   replace_scc, &  ! in string replace char by char
                   replace_css, &  ! in char replace string by string
                   replace_csc, &  ! in char replace string by char
                   replace_ccs, &  ! in char replace char by string
                   replace_ccc     ! in char replace char by char
ENDINTERFACE

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

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

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

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

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

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

INTERFACE TRIM
  MODULE PROCEDURE trim_s
ENDINTERFACE

INTERFACE IACHAR
  MODULE PROCEDURE iachar_s
ENDINTERFACE

INTERFACE ICHAR
  MODULE PROCEDURE ichar_s
ENDINTERFACE
```

```
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=*),INTENT(IN) :: 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      !----- ASSIGNMENT Procedures ----------------------------------------------!
```

```
 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 assignement
  !  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
  DO i = 1,lb
    s_concat_c%chars(la+i) = string_b(i:i)
  ENDDO
 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(i) = string_a(i:i)
    ENDDO
    c_concat_s%chars(1+la:la+lb) = string_b%chars
  ENDFUNCTION c_concat_s

!----- Reapeated concatenation procedures ---------------------------------!
FUNCTION repeat_s(string,ncopies)
  type(VARYING_STRING),INTENT(IN) :: string
  INTEGER,INTENT(IN)              :: ncopies
  type(VARYING_STRING)            :: repeat_s
  ! 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
    type(VARYING_STRING),INTENT(IN) :: string_a
    CHARACTER(LEN=*),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
```

```
    c_eq_s = .FALSE.; RETURN
  ELSEIF( la < lb .AND. ANY( blank /= string_b%chars(la+1:lb) ) )THEN
    c_eq_s = .FALSE.; RETURN
  ENDIF
  c_eq_s = .TRUE.
 ENDFUNCTION 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
    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(i:i) )THEN
      s_lt_c = .TRUE.; RETURN
    ELSEIF( string_a%chars(i) > string_b(i:i) )THEN
      s_lt_c = .FALSE.; RETURN
    ENDIF
  ENDDO
  IF( la < lb )THEN
    IF( blank < string_b(la+1:lb) )THEN
      s_lt_c = .TRUE.; RETURN
    ELSEIF( blank > string_b(la+1:lb) )THEN
      s_lt_c = .FALSE.; RETURN
    ENDIF
  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(i:i) < string_b%chars(i) )THEN
      c_lt_s = .TRUE.; RETURN
    ELSEIF( string_a(i:i) > string_b%chars(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
    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 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

   !------ Greater-than-or-equal-to operators ---------------------------------!
    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%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
  IF( la < lb )THEN
    DO i = la+1,lb
      IF( blank > string_b%chars(i) )THEN
        c_ge_s = .TRUE.; RETURN
      ELSEIF( blank < string_b%chars(i) )THEN
        c_ge_s = .FALSE.; RETURN
      ENDIF
    ENDDO
  ELSEIF( la > lb )THEN
    IF( string_a(lb+1:la) > blank )THEN
      c_ge_s = .TRUE.; RETURN
    ELSEIF( string_a(lb+1:la) < blank )THEN
      c_ge_s = .FALSE.; RETURN
    ENDIF
  ENDIF
  c_ge_s = .TRUE.
 ENDFUNCTION c_ge_s

!------ 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
```

```
  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
  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
  IF( 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
  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
  IF( 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
  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
  s_llt_c = .FALSE.
ENDFUNCTION s_llt_c

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
  c_llt_s = .FALSE.
ENDFUNCTION c_llt_s

!----- LLE procedures -----------------------------------------------------!
FUNCTION s_lle_s(string_a,string_b)  ! string_a<=string_b ISO-646 ordering
 type(VARYING_STRING),INTENT(IN) :: string_a,string_b
 LOGICAL                         :: s_lle_s
 ! Returns TRUE if strings are equal or if string_a preceeds string_b in the
 ! ISO 646 collating sequence.  Otherwise the result is FALSE.
 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
    DO i = lb+1,la
      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
    ENDDO
  ENDIF
  s_lle_s = .TRUE.
ENDFUNCTION s_lle_s

FUNCTION s_lle_c(string_a,string_b)  ! strung_a<=string_b ISO-646 ordering
```

```
 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
    IF( LLT(blank,string_b(la+1:lb)) )THEN
      s_lle_c = .TRUE.; RETURN
    ELSEIF( LGT(blank,string_b(la+1:lb)) )THEN
      s_lle_c = .FALSE.; RETURN
    ENDIF
  ELSEIF( la > lb )THEN
    DO i = lb+1,la
      IF( LLT(string_a%chars(i),blank) )THEN
        s_lle_c = .TRUE.; RETURN
      ELSEIF( LGT(string_a%chars(i),blank) )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
  ELSEIF( la > lb )THEN
    IF( LLT(string_a(lb+1:la),blank) )THEN
      c_lle_s = .TRUE.; RETURN
    ELSEIF( LGT(string_a(lb+1:la),blank) )THEN
      c_lle_s = .FALSE.; RETURN
    ENDIF
  ENDIF
  c_lle_s = .TRUE.
ENDFUNCTION c_lle_s

!----- LGE procedures -----------------------------------------------------!
FUNCTION s_lge_s(string_a,string_b)  ! string_a>=string_b ISO-646 ordering
 type(VARYING_STRING),INTENT(IN) :: string_a,string_b
 LOGICAL                         :: s_lge_s
 ! Returns TRUE if strings are equal or if string_a follows string_b in the
 ! ISO 646 collating sequence. Otherwise the result is FALSE.
 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(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
    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
    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
    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
  c_lge_s = .TRUE.
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
  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
    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
 type(VARYING_STRING),INTENT(IN) :: string_b
 LOGICAL                         :: c_lgt_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_lgt_s = .TRUE.; RETURN
    ELSEIF( LLT(string_a(i:i),string_b%chars(i)) )THEN
      c_lgt_s = .FALSE.; RETURN
    ENDIF
  ENDDO
  IF( la < lb )THEN
    DO i = la+1,lb
      IF( LGT(blank,string_b%chars(i)) )THEN
        c_lgt_s = .TRUE.; RETURN
      ELSEIF( LLT(blank,string_b%chars(i)) )THEN
        c_lgt_s = .FALSE.; RETURN
      ENDIF
    ENDDO
  ELSEIF( la > lb )THEN
    IF( LGT(string_a(lb+1:la),blank) )THEN
      c_lgt_s = .TRUE.; RETURN
    ELSEIF( LLT(string_a(lb+1:la),blank) )THEN
      c_lgt_s = .FALSE.; RETURN
    ENDIF
  ENDIF
  c_lgt_s = .FALSE.
ENDFUNCTION c_lgt_s


!----- Input string procedure ----------------------------------------------!
SUBROUTINE get_d_eor(string,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
 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
```

```
      iostat=ist
      RETURN
    ELSE
      WRITE(*,*) " Error No.",ist, &
                 " during READ_STRING of varying string on default 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_d_eor

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

```
 type(VARYING_STRING),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, occurance 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(*,FMT='(A)',ADVANCE='NO',EOR=9999,IOSTAT=ist) buffer
   IF( ist /= 0 )THEN
     IF(PRESENT(iostat)) THEN
       iostat=ist
       RETURN
     ELSE
       WRITE(*,*) " Error No.",ist, &
                  " during READ_STRING of varying string on default unit"
       STOP
     ENDIF
   ENDIF
   ! check for occurance 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_d_tset_s

SUBROUTINE get_u_tset_s(unit,string,set,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
 type(VARYING_STRING),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 unit starting at next character in the file and
```

```
1       ! terminating at the end of record, occurance of a character in set,
2       ! or after reading maxlen characters.
3        CHARACTER :: buffer  ! characters must be read one at a time to detect
4                             ! first terminator character in set
5        INTEGER   :: ist,toread,lenset
6        ist=0
7        lenset = LEN(set)
8        IF(PRESENT(maxlen))THEN
9          toread=maxlen
10       ELSE
11         toread=HUGE(1)
12       ENDIF
13       string = ""  ! clears return string
14       DO  ! repeatedly read buffer and add to string until EoR
15           ! or maxlen reached
16         IF(toread <= 0)EXIT
17         READ(unit,FMT='(A)',ADVANCE='NO',EOR=9999,IOSTAT=ist) buffer
18         IF( ist /= 0 )THEN
19           IF(PRESENT(iostat)) THEN
20             iostat=ist
21             RETURN
22           ELSE
23             WRITE(*,*) " Error No.",ist, &
24                        " during READ_STRING of varying string on unit ",unit
25             STOP
26           ENDIF
27         ENDIF
28         ! check for occurance of set character in buffer
29           DO j = 1,lenset
30             IF(buffer == set%chars(j)) GOTO 9999
31           ENDDO
32         string = string//buffer
33         toread = toread - 1
34       ENDDO
35       IF(PRESENT(iostat)) iostat = ist
36       RETURN
37       9999 string = string//buffer
38       IF(PRESENT(iostat)) iostat = ist
39      ENDSUBROUTINE get_u_tset_s

40      SUBROUTINE get_d_tset_c(string,set,maxlen,iostat)
41       type(VARYING_STRING),INTENT(OUT) :: string
42                                        ! the string variable to be filled with
43                                        ! characters read from the
44                                        ! file connected to the default unit
45       CHARACTER(LEN=*),INTENT(IN)      :: set
46                                        ! the set of characters which if found in
47                                        ! the input terminate the read
48       INTEGER,INTENT(IN),OPTIONAL      :: maxlen
49                                        ! if present indicates the maximum
50                                        ! number of characters that will be
51                                        ! read from the file
52       INTEGER,INTENT(OUT),OPTIONAL     :: iostat
53                                        ! if present used to return the status
54                                        ! of the data transfer
55                                        ! if absent errors cause termination
56      ! reads string from the default unit starting at next character in the file and
57      ! terminating at the end of record, occurance of a character in set,
58      ! or after reading maxlen characters.
59       CHARACTER :: buffer  ! characters must be read one at a time to detect
60                            ! first terminator character in set
61       INTEGER   :: ist,toread,lenset
62       ist=0
63       lenset = LEN(set)
64       IF(PRESENT(maxlen))THEN
65         toread=maxlen
66       ELSE
67         toread=HUGE(1)
68       ENDIF
69       string = ""  ! clears return string
70       DO  ! repeatedly read buffer and add to string until EoR
71           ! or maxlen reached
```

```
   IF(toread <= 0)EXIT
   READ(*,FMT='(A)',ADVANCE='NO',EOR=9999,IOSTAT=ist) buffer
   IF( ist /= 0 )THEN
     IF(PRESENT(iostat)) THEN
       iostat=ist
       RETURN
     ELSE
       WRITE(*,*) " Error No.",ist, &
                  " during READ_STRING of varying string on default unit"
       STOP
     ENDIF
   ENDIF
   ! check for occurance 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

SUBROUTINE get_u_tset_c(unit,string,set,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
 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 unit starting at next character in the file and
! terminating at the end of record, occurance 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
     IF(PRESENT(iostat)) THEN
       iostat=ist
       RETURN
     ELSE
       WRITE(*,*) " Error No.",ist, &
                  " during READ_STRING of varying string on unit ",unit
       STOP
     ENDIF
   ENDIF
```

```
   ! check for occurance 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
                                  ! the string 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) 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
                                  ! 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
  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
                                  ! 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( 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
 ENDIF
 IF(PRESENT(iostat)) iostat=0
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
                                    ! 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
! 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(*,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
  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
! 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) 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=*),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
                                  ! 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
! 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(*,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=*),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
! 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 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, ie. 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=*),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, ie. 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=*),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 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, ie. 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
  work(ip:ip+lsub-1) =substring%chars
  DO i=ip,ls
    work(i+lsub) = string(i:i)
  ENDDO
  insert_cs%chars => work
 ENDFUNCTION insert_cs

 FUNCTION insert_cc(string,start,substring)
  type(VARYING_STRING)        :: insert_cc
  CHARACTER(LEN=*),INTENT(IN) :: string
  INTEGER,INTENT(IN)          :: start
  CHARACTER(LEN=*),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, ie. 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_cc%chars => work
 ENDFUNCTION insert_cc

!----- Replace procedures -----------------------------------------------------!
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)
 nw = MAX(ls,ip+lsub-1)
 ALLOCATE(work(1:nw))
```

```
 work(1:ip-1) = string%chars(1:ip-1)
 work(ip:ip+lsub-1) = substring%chars
 work(ip+lsub:nw) = string%chars(ip+lsub:ls)
 replace_ss%chars => work
ENDFUNCTION replace_ss

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 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))
 work(1:ip-1) = string%chars(1:ip-1)
 work(ip:ip+lsub-1) = substring%chars
 work(ip+lsub:nw) = string%chars(if+1: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=*),INTENT(IN)     :: substring
 !  calculates the result string by the following actions:
 !  inserts the characters from 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)
 nw = MAX(ls,ip+lsub-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_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=*),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
```

```
 1        CHARACTER,POINTER,DIMENSION(:) :: work
 2        INTEGER                        :: ip,is,if,nw,lsub,ls
 3        lsub = LEN(substring); ls = LEN(string)
 4        is = MAX(start,1)
 5        ip = MIN(ls+1,is)
 6        if = MAX(ip-1,MIN(finish,ls))
 7        nw = lsub + ls - if+ip-1
 8        ALLOCATE(work(1:nw))
 9        work(1:ip-1) = string%chars(1:ip-1)
10        DO i = 1,lsub
11          work(ip-1+i) = substring(i:i)
12        ENDDO
13        work(ip+lsub:nw) = string%chars(if+1:ls)
14        replace_sc_sf%chars => work
15      ENDFUNCTION replace_sc_sf

16      FUNCTION replace_cs(string,start,substring)
17        type(VARYING_STRING)            :: replace_cs
18        CHARACTER(LEN=*),INTENT(IN)     :: string
19        INTEGER,INTENT(IN)              :: start
20        type(VARYING_STRING),INTENT(IN) :: substring
21        !  calculates the result string by the following actions:
22        !  inserts the substring into string beginning at position
23        !  start replacing the following LEN(substring) characters of the string
24        !  and enlarging string if necessary. if start is greater than LEN(string)
25        !  substring is simply appended to string by concatenation. If start is less
26        !  than 1, substring replaces characters in string starting at 1
27        CHARACTER,POINTER,DIMENSION(:) :: work
28        INTEGER                        :: ip,is,nw,lsub,ls
29        lsub = LEN(substring); ls = LEN(string)
30        is = MAX(start,1)
31        ip = MIN(ls+1,is)
32        nw = MAX(ls,ip+lsub-1)
33        ALLOCATE(work(1:nw))
34        DO i=1,ip-1
35          work(i) = string(i:i)
36        ENDDO
37        work(ip:ip+lsub-1) = substring%chars
38        DO i=ip+lsub,nw
39          work(i) = string(i:i)
40        ENDDO
41        replace_cs%chars => work
42      ENDFUNCTION replace_cs
43
44      FUNCTION replace_cs_sf(string,start,finish,substring)
45        type(VARYING_STRING)            :: replace_cs_sf
46        CHARACTER(LEN=*),INTENT(IN)     :: string
47        INTEGER,INTENT(IN)              :: start,finish
48        type(VARYING_STRING),INTENT(IN) :: substring
49        !  calculates the result string by the following actions:
50        !  inserts the substring into string beginning at position
51        !  start replacing the following finish-start+1 characters of the string
52        !  and enlarging or shrinking the string if necessary.
53        !  If start is greater than LEN(string) substring is simply appended to string
54        !  by concatenation. If start is less than 1, start = 1 is used
55        !  If finish is greater than LEN(string), finish = LEN(string) is used
56        !  If finish is less than start, substring is inserted before start
57        CHARACTER,POINTER,DIMENSION(:) :: work
58        INTEGER                        :: ip,is,if,nw,lsub,ls
59        lsub = LEN(substring); ls = LEN(string)
60        is = MAX(start,1)
61        ip = MIN(ls+1,is)
62        if = MAX(ip-1,MIN(finish,ls))
63        nw = lsub + ls - if+ip-1
64        ALLOCATE(work(1:nw))
65        DO i=1,ip-1
66          work(i) = string(i:i)
67        ENDDO
68        work(ip:ip+lsub-1) = substring%chars
69        DO i=1,nw-ip-lsub+1
70          work(i+ip+lsub-1) = string(if+i:if+i)
71        ENDDO
```

```
 replace_cs_sf%chars => work
ENDFUNCTION replace_cs_sf

FUNCTION replace_cc(string,start,substring)
 type(VARYING_STRING)            :: replace_cc
 CHARACTER(LEN=*),INTENT(IN)     :: string
 INTEGER,INTENT(IN)              :: start
 CHARACTER(LEN=*),INTENT(IN)     :: substring
 !  calculates the result string by the following actions:
 !  inserts the characters from 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)
 nw = MAX(ls,ip+lsub-1)
 ALLOCATE(work(1:nw))
 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+lsub,nw
   work(i) = string(i:i)
 ENDDO
 replace_cc%chars => work
ENDFUNCTION replace_cc

FUNCTION replace_cc_sf(string,start,finish,substring)
 type(VARYING_STRING)            :: replace_cc_sf
 CHARACTER(LEN=*),INTENT(IN)     :: string
 INTEGER,INTENT(IN)              :: start,finish
 CHARACTER(LEN=*),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
 DO i=1,lsub
   work(i+ip-1) = substring(i:i)
 ENDDO
 DO i=1,nw-ip-lsub+1
   work(i+ip+lsub-1) = string(if+i:if+i)
 ENDDO
 replace_cc_sf%chars => work
ENDFUNCTION replace_cc_sf

FUNCTION replace_sss(string,target,substring,every,back)
 type(VARYING_STRING)            :: replace_sss
 type(VARYING_STRING),INTENT(IN) :: string,target,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 occurences
!  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 occurance 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 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: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
    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(string%chars(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(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
    ENDIF
    ipos=ipos+1; ipow=ipow+1
  ENDDO
ENDIF
replace_sss%chars => work
ENDFUNCTION replace_sss

FUNCTION replace_ssc(string,target,substring,every,back)
 type(VARYING_STRING)           :: replace_ssc
 type(VARYING_STRING),INTENT(IN) :: string,target
```

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

```
   ENDDO
 ENDIF
 replace_ssc%chars => work
ENDFUNCTION replace_ssc

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 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 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
     ! 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: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
     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(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: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
      ENDIF
      ipos=ipos+1; ipow=ipow+1
    ENDDO
  ENDIF
  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 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 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
      ! 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: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 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: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
      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=*),INTENT(IN)     :: string
  type(VARYING_STRING),INTENT(IN) :: target,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 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,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 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(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
      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(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
      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
  type(VARYING_STRING),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 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,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 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(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 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(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
      ENDIF
      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 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 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
  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.
```

```
 1        ENDIF
 2        IF( PRESENT(every) )THEN
 3          rep_search = every
 4        ELSE
 5          rep_search = .FALSE.
 6        ENDIF
 7        IF( dir_switch )THEN ! backwards search
 8          ipos = ls-lt+1
 9          DO
10            IF( ipos < 1 )EXIT ! search past start of string
11            ! test for occurance of target in string at this position
12            IF( string(ipos:ipos+lt-1) == target )THEN
13              ! match found allocate space for string with this occurance of
14              ! target replaced by substring
15              ALLOCATE(temp(1:SIZE(work)+lsub-lt))
16              ! copy work into temp replacing this occurance of target by
17              ! substring
18              temp(1:ipos-1) = work(1:ipos-1)
19              temp(ipos:ipos+lsub-1) = substring%chars
20              temp(ipos+lsub:) = work(ipos+lt:)
21              work => temp ! make new version of work point at the temp space
22              IF(.NOT.rep_search)EXIT ! exit if only first replacement wanted
23              ! move search and replacement positions over the effected positions
24              ipos = ipos-lt+1
25            ENDIF
26            ipos=ipos-1
27          ENDDO
28        ELSE ! forward search
29          ipos = 1; ipow = 1
30          DO
31            IF( ipos > ls-lt+1 )EXIT ! search past end of string
32            ! test for occurance of target in string at this position
33            IF( string(ipos:ipos+lt-1) == target )THEN
34              ! match found allocate space for string with this occurance of
35              ! target replaced by substring
36              ALLOCATE(temp(1:SIZE(work)+lsub-lt))
37              ! copy work into temp replacing this occurance of target by
38              ! substring
39              temp(1:ipow-1) = work(1:ipow-1)
40              temp(ipow:ipow+lsub-1) = substring%chars
41              temp(ipow+lsub:) = work(ipow+lt:)
42              work => temp ! make new version of work point at the temp space
43              IF(.NOT.rep_search)EXIT ! exit if only first replacement wanted
44              ! move search and replacement positions over the effected positions
45              ipos = ipos+lt-1; ipow = ipow+lsub-1
46            ENDIF
47            ipos=ipos+1; ipow=ipow+1
48          ENDDO
49        ENDIF
50        replace_ccs%chars => work
51      ENDFUNCTION replace_ccs

52      FUNCTION replace_ccc(string,target,substring,every,back)
53        type(VARYING_STRING)            :: replace_ccc
54        CHARACTER(LEN=*),INTENT(IN)     :: string,target,substring
55        LOGICAL,INTENT(IN),OPTIONAL     :: every,back
56        !  calculates the result string by the following actions:
57        !  searches for occurences of target in string, and replaces these with
58        !  substring. if back present with value true search is backward otherwise
59        !  search is done forward. if every present with value true all accurences
60        !  of target in string are replaced, otherwise only the first found is
61        !  replaced. if target is not found the result is the same as string.
62        LOGICAL                         :: dir_switch, rep_search
63        CHARACTER,DIMENSION(:),POINTER :: work,temp
64        INTEGER                         :: ls,lt,lsub,ipos,ipow
65        ls = LEN(string); lt = LEN(target); lsub = LEN(substring)
66        IF(lt==0)THEN
67          WRITE(*,*) " Zero length target in REPLACE"
68          STOP
69        ENDIF
70        ALLOCATE(work(1:ls))
71        DO i=1,ls
```

```
      work(i) = string(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( string(ipos:ipos+lt-1) == target )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: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 occurance of target in string at this position
        IF( string(ipos:ipos+lt-1) == target )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: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
        ENDIF
        ipos=ipos+1; ipow=ipow+1
      ENDDO
    ENDIF
    replace_ccc%chars => work
  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=*),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))
   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

!----- Extract procedures --------------------------------------------------!
 FUNCTION extract_s(string,start,finish)
  type(VARYING_STRING),INTENT(IN) :: string
  INTEGER,INTENT(IN),OPTIONAL     :: start
  INTEGER,INTENT(IN),OPTIONAL     :: finish
  type(VARYING_STRING)            :: extract_s
  ! extracts the characters between start and finish from 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_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
```

```
      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
 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
  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,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
    ENDIF
    string%chars => string%chars(1:tpos-1)
  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
```

```
    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

 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
```

```
  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
      ENDIF
    ENDDO
    scan_ss = 0
  ELSE ! forward search
    DO i = 1,ls
tmp=string%chars(i) ! bug work round
      IF( ANY( set%chars == tmp ) )THEN
        scan_ss = i
        RETURN
      ENDIF
    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
  ! 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
        ENDIF
      ENDDO
      IF( matched )THEN
        scan_sc = i
        RETURN
      ENDIF
    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
        ENDIF
      ENDDO
      IF( matched )THEN
        scan_sc = i
        RETURN
      ENDIF
    ENDDO
```

```
      scan_sc = 0
    ENDIF
   ENDFUNCTION scan_sc

   FUNCTION scan_cs(string,set,back)
    CHARACTER(LEN=*),INTENT(IN)     :: string
    type(VARYING_STRING),INTENT(IN) :: set
    LOGICAL,INTENT(IN),OPTIONAL     :: back
    INTEGER                         :: 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
          ENDIF
        ENDDO
        IF( matched )THEN
          scan_cs = i
          RETURN
        ENDIF
      ENDDO
      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
          ENDIF
        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),INTENT(IN) :: string,set
    LOGICAL,INTENT(IN),OPTIONAL     :: back
    INTEGER                         :: 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
  CHARACTER::tmp ! F90 1.1 bug work round
    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
```

```
        verify_ss = i
        RETURN
      ENDIF
    ENDDO
    verify_ss = 0
  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
    verify_ss = 0
  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
        ! cycle string search if string character found in set
      ENDDO
      ! string character not found in set index i is result
        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
    verify_sc = 0
  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 icharacter 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(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.

```
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
                       ! input unit assumed to be the whole of the record read
OPEN(UNIT=1,FILE=CHAR(fname))  ! CHAR(fname) converts to the type
                               ! required by FILE= specifier
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)
      EXIT word_scan
    ENDIF
    IF(nd > 1) wcount=wcount+1  ! at least one non-terminator character
                                ! in the word
    line = REMOVE(line,1,nd)  ! strips the counted word and its terminator
                              ! from the line reducing its length before
                              ! rescanning for the next word
  ENDDO word_scan
  IF(nd > 0) wcount=wcount+1
ENDDO file_read
IF(ierr < 0)THEN
  WRITE(*,*) "No. of words in file =",wcount
ELSEIF(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.

```
PROGRAM vocabulary_word_count
!------------------------------------------------------------------------------!
! Counts the number of "words" contained in a file. The words are assumed to  !
! be terminated by any one of:                                                 !
```

```
1       ! space,comma,period,!,?, or the EoR                                        !
2       ! The file may have records of any length and the file may contain any number !
3       ! of records.                                                               !
4       ! The program prompts for the name of the file to be subject to a word count  !
5       ! and the result is written to the default output unit                      !
6       ! Also builds a list of the vocabulary found and the frequency of occurance   !
7       ! of each different word.                                                   !
8       !--------------------------------------------------------------------------!
9       USE ISO_VARYING_STRING
10      type(VARYING_STRING) :: line,word,fname
11      INTEGER              :: ierr,nd,wcount=0
12      !--------------------------------------------------------------------------!
13      ! Vocabulary list and frequency count arrays. The size of these arrays will   !
14      ! be extended dynamically in steps of 100 as the used vocabulary grows        !
15      !--------------------------------------------------------------------------!
16      type(VARYING_STRING),ALLOCATABLE,DIMENSION(:) :: vocab
17      INTEGER,ALLOCATABLE,DIMENSION(:)              :: freq
18      INTEGER                                       :: list_size=100,list_top=0
19      !--------------------------------------------------------------------------!
20      ! Initialise the lists and determine the file to be processed                !
21      !--------------------------------------------------------------------------!
22      ALLOCATE(vocab(1:list_size),freq(1:list_size))
23      WRITE(*,ADVANCE='NO',FMT='(A)') " Input name of file?"
24      CALL GET(STRING=fname)  ! read the required filename from the default
25                              ! input unit assumed to be the whole of the record read
26      OPEN(UNIT=1,FILE=CHAR(fname))  ! CHAR(fname) converts to the type
27                                     ! required by FILE= specifier
28      file_read: DO ! until EoF reached
29        CALL GET(1,line,IOSTAT=ierr)  ! read next line of file
30        IF(ierr == -1)EXIT file_read
31        word_scan: DO ! until end of line
32          nd=SCAN(line," ,.!?")  ! scan to find end of word
33          IF(nd == 0)THEN  ! EoR is end of word
34            nd = LEN(line)
35            EXIT word_scan
36          ENDIF
37          IF(nd > 1)THEN  ! at least one non-terminator character in the word
38             wcount=wcount+1
39             word = EXTRACT(line,1,nd-1)
40             CALL update_vocab_lists
41          ENDIF
42          line = REMOVE(line,1,nd)  ! strips the counted word and its terminator
43                                    ! from the line reducing its length before
44                                    ! rescanning for the next word
45        ENDDO word_scan
46        IF(nd > 0)THEN  ! at least one character in the word
47           wcount=wcount+1
48           word = EXTRACT(line,1,nd-1)
49           CALL update_vocab_lists
50         ENDIF
51      ENDDO file_read
52      IF(ierr < 0)THEN
53        WRITE(*,*) "No. of words in file =",wcount
54        WRITE(*,*) "There are ",list_top,"  distinct words"
55        WRITE(*,*) "with the following frequencies of occurance"
56        print_loop: DO i=1,list_top
57          WRITE(*,FMT='(1X,I6,2X)',ADVANCE='NO') freq(i)
58          CALL PUT_LINE(STRING=vocab(i))
59        ENDDO print_loop
60      ELSEIF(ierr > 0)THEN
61        WRITE(*,*) "Error in GET in vocabulary_word_count, No.",ierr
62      ENDIF

63      CONTAINS

64      SUBROUTINE extend_lists
65      !--------------------------------------------------------------------------!
66      ! Accesses the host variables:                                             !
67      !  type(VARYING_STRING),ALLOCATABLE,DIMENSION(:) :: vocab                  !
68      !  INTEGER,ALLOCATABLE,DIMENSION(:)              :: freq                   !
69      !  INTEGER                                       :: list_size              !
70      ! so as to extend the size of the lists preserving the existing vocabulary  !
```

```
1       ! and frequency information in the new extended lists                        !
2       !-----------------------------------------------------------------------------!
3       type(VARYING_STRING),DIMENSION(list_size) :: vocab_swap
4       INTEGER,DIMENSION(list_size)              :: freq_swap
5       INTEGER,PARAMETER :: list_increment=100
6       INTEGER           :: new_list_size,alerr
7       vocab_swap = vocab  ! copy old list into temporary space
8       freq_swap =freq
9       new_list_size = list_size + list_increment
10      DEALLOCATE(vocab,freq)
11      ALLOCATE(vocab(1:new_list_size),freq(1:new_list_size),STAT=alerr)
12      IF(alerr /= 0)THEN
13        WRITE(*,*) "Unable to extend vocabulary list"
14        STOP
15      ENDIF
16      vocab(1:list_size) = vocab_swap   ! copy old list back into bottom
17      freq(1:list_size) = freq_swap     ! of new extended list
18      list_size = new_list_size
19      ENDSUBROUTINE extend_lists

20      SUBROUTINE update_vocab_lists
21      !-----------------------------------------------------------------------------!
22      ! Accesses the host variables:                                                !
23      !  type(VARYING_STRING),ALLOCATABLE,DIMENSION(:) :: vocab                     !
24      !  INTEGER,ALLOCATABLE,DIMENSION(:)              :: freq                      !
25      !  INTEGER                                       :: list_size,list_top        !
26      !  type(VARYING_STRING)                          :: word                      !
27      ! searches the existing words in vocab to find a match for word               !
28      ! if found increments the freq if not found adds word to                      !
29      ! list_top + 1  vocab list and sets corresponding freq to 1                   !
30      ! if list_size exceeded extend the list size before updating                  !
31      !-----------------------------------------------------------------------------!
32      list_search: DO i=1,list_top
33        IF(word == vocab(i))THEN
34          freq(i) = freq(i) + 1
35          RETURN
36        ENDIF
37      ENDDO list_search
38      IF(list_top == list_size)THEN
39        CALL extend_lists
40      ENDIF
41      list_top = list_top + 1
42      vocab(list_top) = word
43      freq(list_top) = 1
44      ENDSUBROUTINE update_vocab_lists

45      ENDPROGRAM vocabulary_word_count
```