
**Information technology — Programming
languages — Fortran —**

Part 2:

Varying length character strings

*Technologies de l'information — Langages de programmation —
Fortran —*

Partie 2: Chaînes de caractères de longueur variable

Contents

Foreword	iii
Introduction	iv
Section 1 : General	1
1.1 Scope	1
1.2 Normative References	2
Section 2 : Requirements	3
2.1 The Name of the Module	3
2.2 The Type	3
2.3 Extended Meanings for Intrinsic Operators	3
2.3.1 Assignment	3
2.3.2 Concatenation	4
2.3.3 Comparisons	4
2.4 Extended Meanings for Generic Intrinsic Procedures	4
2.4.1 The LEN procedure	4
2.4.2 The CHAR procedure	4
2.4.3 The ICHAR procedure	5
2.4.4 The IACHAR procedure	5
2.4.5 The TRIM procedure	5
2.4.6 The LEN_TRIM procedure	6
2.4.7 The ADJUSTL procedure	6
2.4.8 The ADJUSTR procedure	6
2.4.9 The REPEAT procedure	6
2.4.10 Comparison procedures	7
2.4.11 The INDEX procedure	7
2.4.12 The SCAN procedure	7
2.4.13 The VERIFY procedure	8
2.5 Additional Generic Procedure for Type Conversion	8
2.5.1 The VAR_STR procedure	8
2.6 Additional Generic Procedures for Input/Output	9
2.6.1 The GET procedure	9
2.6.2 The PUT procedure	10
2.6.3 The PUT_LINE procedure	10
2.7 Additional Generic Procedures for Substring Manipulation	11
2.7.1 The INSERT procedure	11
2.7.2 The REPLACE procedure	11
2.7.3 The REMOVE procedure	12
2.7.4 The EXTRACT procedure	13
2.7.5 The SPLIT procedure	13
Annex A : Module ISO_varying_string	14
Annex B : Examples	64

© ISO/IEC 1994

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from the publisher.

ISO/IEC Copyright Office • Case postale 56 • CH-1211 Genève 20 • Switzerland

Printed in Switzerland

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

International Standard ISO/IEC 1539-2 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*.

ISO/IEC 1539 consists of the following parts, under the general title *Information technology — Programming languages — Fortran*:

- *Part 1: Core Language Fortran*
- *Part 2: Varying length character strings*

Annexes A and B of this part of ISO/IEC 1539 are for information only.

Introduction

This part of ISO/IEC 1539 has been prepared by ISO/IEC JTC1/SC22/WG5, the technical working group for the Fortran language. This part of ISO/IEC 1539 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 part of ISO/IEC 1539 is the second part. The revised language defined by the above standard is informally known as Fortran 90.

This part of ISO/IEC 1539 defines the interface and semantics for a module that provides facilities for the manipulation of character strings of arbitrary and dynamically variable length. Annex A includes a possible implementation, in Fortran 90, of a module that conforms to this part of ISO/IEC 1539. It should be noted, however, that this is purely for purposes of demonstrating the feasibility and portability of this standard. The actual code shown in this annex is not intended in any way to prescribe the method of implementation, nor is there any implication that this is in any way an optimal portable implementation. The module is merely a fairly straightforward demonstration that a portable implementation is possible.

IECNORM.COM : Click to view the full PDF of ISO/IEC 1539-2:1994

Information technology — Programming languages — Fortran —

Part 2:

Varying length character strings

Section 1: General

1.1 Scope

This part of ISO/IEC 1539 defines facilities for use in Fortran for the manipulation of character strings of dynamically variable length. This part of ISO/IEC 1539 provides an auxiliary standard for the version of the Fortran language informally known as Fortran 90. The International Standard defining this revision of the Fortran language is

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

This part of ISO/IEC 1539 is an auxiliary standard to that defining Fortran 90 in that it defines additional facilities to those defined intrinsically in the primary language standard. A processor conforming to the Fortran 90 standard is not required to also conform to this part of ISO/IEC 1539. However, conformance to this part of ISO/IEC 1539 assumes conformance to the primary Fortran 90 standard.

This part of ISO/IEC 1539 prescribes the name of a Fortran module, the name of a derived data type to be used to represent varying-length strings, the interfaces for the procedures and operators to 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 part of ISO/IEC 1539 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 part of ISO/IEC 1539 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 part of ISO/IEC 1539, and that the semantics are as required by this part of ISO/IEC 1539 and those of ISO/IEC 1539 : 1991.

It should be noted that a processor is not required to implement this part of ISO/IEC 1539 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 part of ISO/IEC 1539.

A processor conforming to this part of ISO/IEC 1539 may extend the facilities provided for the manipulation of varying length character strings as long as such extensions do not conflict with this part of ISO/IEC 1539 or with ISO/IEC 1539 : 1991.

A module, written in standard conforming Fortran, is included in Annex A. This module illustrates one way in which the facilities described in this part of ISO/IEC 1539 could be provided. This module is both conformant with the requirements of this part of ISO/IEC 1539 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 part of ISO/IEC 1539 defines facilities for dynamically varying length strings of characters of default kind only. Throughout this part of ISO/IEC 1539 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 part of ISO/IEC 1539 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 that apply to fixed length character strings have extended meanings defined by this part of ISO/IEC 1539 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.

1.2 Normative references

The following standards contain provisions which, through reference in this text, constitute provisions of this part of ISO/IEC 1539. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this part of ISO/IEC 1539 are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below. Members of IEC and ISO maintain registers of currently valid International Standards.

ISO/IEC 646:1991, *Information technology — ISO 7-bit coded character set for information interchange*.

ISO/IEC 1539:1991, *Information technology — Programming languages — Fortran*.

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 part of ISO/IEC 1539 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 that 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.**, and **.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 Attributes: scalar of 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 ==, /=, <, <=, >=, or > shall be defined for scalar operands with the following type combinations:

VARYING_STRING .OP. **VARYING_STRING**,

VARYING_STRING .OP. **CHARACTER**, or

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.**, and **.GT.** also have their meanings extended in this manner.

Result Attributes: scalar of type default **LOGICAL**.

Result Value: The result value is true if **string_a** stands in the indicated relation to **string_b** and is false otherwise. 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**. The results produced in each case are also scalar.

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.

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

Result Attributes: scalar of 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.

Arguments:

string - is of type **VARYING_STRING**
length - is of type default **INTEGER**.

The arguments are scalars and are unchanged by the procedure.

Result Attributes: scalar of type default **CHARACTER**. If **length** is absent, the result has the same length as **string**. If **length** is present, the result has the length specified by the argument **length**.

Result Value:

- Case(i) : If **length** is absent, the result is a copy of the characters in the argument **string**.
 Case(ii) : If **length** is present, the result is a copy of the characters in the argument **string** that may have been truncated or padded. 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: returns the position of a character in the processor defined collating sequence.

Argument: **c** is a scalar of type **VARYING_STRING** and of length exactly one. The argument is unchanged by the procedure.

Result Attributes: scalar of type default **INTEGER**.

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

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

IACHAR(c)

shall be added.

Description: returns the position of a character in the collating sequence defined by the International Standard ISO 646 : 1991.

Argument: **c** is a scalar of type **VARYING_STRING** and of length exactly one. The argument is unchanged by the procedure.

Result Attributes: scalar of type default **INTEGER**.

Result Value: The result value is the position of the character **c** in the collating sequence defined by the International Standard ISO 646 : 1991 for default characters. If the character **c** is not defined in the standard set, the result is processor dependent but is always equal to **IACHAR(CHAR(c))**.

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

TRIM(string)

shall be added.

Description: removes trailing blanks from a string.

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

Result Attributes: scalar of type **VARYING_STRING**.

Result Value: The result value is the same as **string** except that any trailing blanks have been deleted. 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: returns the length of a string not counting any trailing blanks.**Argument:** **string** is a scalar of type **VARYING_STRING**. The argument is unchanged by the procedure.**Result Attributes:** scalar of 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.**Argument:** **string** is a scalar of type **VARYING_STRING**. The argument is unchanged by the procedure.**Result Attributes:** scalar of type **VARYING_STRING**.**Result Value:** The result value is the same as **string** except that any leading blanks have been deleted and the same number of trailing blanks inserted.**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.**Argument:** **string** is a scalar of type **VARYING_STRING**. The argument is unchanged by the procedure.**Result Attributes:** scalar of type **VARYING_STRING**.**Result Value:** The result value is the same as **string** except that any trailing blanks have been deleted and the same number of leading blanks inserted.**2.4.9 The REPEAT procedure:** The generic function reference of the form**REPEAT(string,ncopies)**

shall be added.

Description: concatenates several copies of a string.**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 Attributes:** scalar of 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**. 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 : 1991 collating sequence.

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 Attributes: scalar of 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 : 1991. 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. If either argument contains a character **c** not defined by the standard, the result value is processor dependent and based on the collating value for **IACHAR(c)**. Zero length strings are considered to be lexically equal.

2.4.11 The INDEX procedure: The generic function reference of the form**INDEX(string, substring, back)**

shall be added.

Description: returns an integer that is the starting position of a substring within a string.

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**.

The arguments are unchanged by the procedure.

Result Attributes: scalar of 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: scans a string for any one of the characters in a set of characters.

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 Attributes: scalar of 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 left-most 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 right-most 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: verifies that a string contains only characters from a given set by scanning for any character not in the set.

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 Attributes: scalar of 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 left-most 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 right-most 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.

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

Result Attributes: scalar of 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,separator,maxlen,iostat)
CALL GET(unit,string,set,separator,maxlen,iostat)
```

shall be provided.

Description: reads characters from an external file into a string.

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**,
- separator** - is of type **VARYING_STRING** and is **OPTIONAL**,
- 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. The arguments **maxlen**, **unit**, and **set** are unchanged by the procedure.

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 argument **separator** is present, the actual character found which terminates the transfer is returned in **separator**. If the transfer is terminated other than by the occurrence of a character in **set**, a zero length string is returned in **separator**. If the transfer is 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 and the end-of-record is not reached, a positive value if an error occurs, and a negative value if an end-of-file or end-of-record condition occurs. Note, the negative value returned for an

end-of-file condition must be different from that returned for an end-of-record condition. If **iostat** is absent and an error or end-of-file condition 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: writes a string to an external file.

Arguments:

string - is either of type **VARYING_STRING** or type **CHARACTER**,
unit - is of type default **INTEGER**,
iostat - is of 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. The arguments **unit** and **string** are unchanged by the procedure.

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: writes a string to an external file and ends the record.

Arguments:

string - is either of type **VARYING_STRING** or type **CHARACTER**
unit - is of type default **INTEGER**
iostat - is of 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. The arguments **unit** and **string** are unchanged by the procedure.

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: inserts a substring into a string at a specified position.

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 are scalars. The arguments are unchanged by the procedure.

Result Attributes: scalar of 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**. 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** and **substring** is inserted before the first character of the copy of **string**.

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

REPLACE(string, 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.

Arguments:

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

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

Result Attributes: scalar of 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 into 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. 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.
- 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, and 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.

Arguments:

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

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

Result Attributes: scalar of type **VARYING_STRING**.

Result Value: The result value is a copy of the characters of **string** with the characters between positions **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.**Arguments:**

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

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

Result Attributes: scalar of type **VARYING_STRING**.

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

2.7.5 The SPLIT procedure: The generic subroutine reference of the form**CALL SPLIT(string, word, set, separator, back)**

shall be provided.

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

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

All arguments are scalar. The arguments **set** and **back** are unchanged by the procedure.

Action: The effect of the procedure is to divide the string at the first occurrence of a character that is 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 or **set** is of zero length, 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)

Module ISO_varying_string

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 part of ISO/IEC 1539. It is not intended to be prescriptive of how facilities consistent with this part of ISO/IEC 1539 should be provided. This module is intended primarily to demonstrate that portable facilities consistent with the interfaces and semantics required by this part of ISO/IEC 1539 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 part of ISO/IEC 1539.

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 part of ISO/IEC 1539 in valid Fortran code, no guarantee is given or implied that this code will produce correct results, or even that it will execute on any particular processor. Neither is there any implication that this illustrative module is in any way an optimal implementation of this standard; it is merely one fairly straightforward portable module that is known to provide a functionally conformant implementation on a few processors.

MODULE ISO_VARYING_STRING

```
! Written by J.L.Schonfelder
! Incorporating suggestions by C.Tanasescu, C.Weber, J.Wagener and W.Walter,
! and corrections due to L.Moss, M.Cohen, P.Griffiths, B.T.Smith
! and many other members of the committee ISO/IEC JTC1/SC22/WG5

! Version produced (5-Jul-94)

!-----!
! This module defines the interface and one possible implementation for a
! dynamic length character string facility in Fortran 90. The Fortran 90
! language is defined by the standard ISO/IEC 1539 : 1991.
! The publicly accessible interface defined by this module is conformant
! with the auxiliary standard, ISO/IEC 1539-2 : 1994.
! The detailed implementation may be considered as an informal definition of
! the required semantics, and may also be used as a guide to the production
! of a portable implementation.
! N.B. Although every care has been taken to produce valid Fortran code in
! construction of this module no guarantee is given or implied that this
! code will work correctly without error on any specific processor, nor
! is this implementation intended to be in any way optimal either in use
! of storage or CPU cycles.
!-----!
```

PRIVATE

```
!-----!
! By default all entities declared or defined in this module are private to
! the module. Only those entities declared explicitly as being public are
! accessible to programs using the module. In particular, the procedures and
! operators defined herein are made accessible via their generic identifiers
! only; their specific names are private.
!-----!
```

TYPE VARYING_STRING

```
PRIVATE
CHARACTER, DIMENSION(:), POINTER :: chars
ENDTYPE VARYING_STRING
```

```
!-----!
! The representation chosen for this definition of the module is of a string
!-----!
```

```

! type consisting of a single component that is a pointer to a rank one array !
! of characters. !
! Note: this Module is defined only for characters of default kind. A similar !
! module could be defined for non-default characters if these are supported !
! on a processor by adding a KIND parameter to the component in the type !
! definition, and to all declarations of objects of CHARACTER type. !
!-----!

CHARACTER, PARAMETER :: blank = " "

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

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

!----- Conversion procedure interfaces -----!
INTERFACE VAR_STR
  MODULE PROCEDURE c_to_s    ! character to string
ENDINTERFACE

INTERFACE CHAR
  MODULE PROCEDURE s_to_c, &    ! string to character
                        s_to_fix_c ! string to specified length character
ENDINTERFACE

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

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

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

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

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

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

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

```

ENDINTERFACE

!----- 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 procedure interfaces -----!
 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 occurrence 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

INTERFACE ADJUSTL
MODULE PROCEDURE adjustl_s
ENDINTERFACE

INTERFACE ADJUSTR
MODULE PROCEDURE adjustr_s
ENDINTERFACE

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

CONTAINS

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

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

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

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

```

```

!----- 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 assignment
  ! reallocates string variable to size of string value and copies characters
  ALLOCATE(var%chars(1:LEN(expr)))
  var%chars = expr%chars
ENDSUBROUTINE s_ass_s

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

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

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

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

!----- Repeated 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
    ENDIF
  ENDDO
  IF( la > lb .AND. string_a(lb+1:la) /= blank )THEN
    s_lt_s = .TRUE.; RETURN
  ELSEIF( la < lb .AND. ANY( blank /= string_b%chars(la+1:lb) ) )THEN
    s_lt_s = .TRUE.; RETURN
  ENDIF
  s_lt_s = .FALSE.
ENDFUNCTION s_lt_s

```

```

    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
        ENDIF
    ENDDO
ENDIF
c_lt_s = .FALSE.
ENDFUNCTION c_lt_s

```

```

        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

```

```

        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

```

```

        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 precedes 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) ! string_a<string_b ISO-646 ordering
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

```

```

        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 precedes 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) ! string_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,separator,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
type(VARYING_STRING), INTENT(OUT), OPTIONAL :: separator
! the actual separator character from set
! found as the input string terminator
! returned as zero length if termination
! by maxlen or EOR
INTEGER, INTENT(IN), OPTIONAL :: maxlen
! if present indicates the maximum
! number of characters that will be
! read from the file
INTEGER, INTENT(OUT), OPTIONAL :: iostat
! if present used to return the status
! of the data transfer
! if absent errors cause termination
! reads string from the default unit starting at next character in the file and
! terminating at the end of record, occurrence of a character in set,
! or after reading maxlen characters.
CHARACTER :: buffer ! characters must be read one at a time to detect
! first terminator character in set
INTEGER :: ist, toread, lenset
lenset = LEN(set)
IF(PRESENT(maxlen)) THEN
  toread=maxlen
ELSE
  toread=HUGE(1)
ENDIF
string = "" ! clears return string
IF(PRESENT(separator)) separator="" ! clear the separator
readchar:DO ! repeatedly read buffer and add to string
  IF(toread <= 0)EXIT readchar ! maxlen reached
  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 GET of varying string on default unit"
      STOP
    ENDIF
  ENDIF
  ! check for occurrence of set character in buffer
  DO j = 1, lenset
    IF(buffer == set%chars(j)) THEN
      IF(PRESENT(separator)) separator=buffer
      EXIT readchar ! separator terminator found
    ENDIF
  ENDDO
  string = string//buffer
  toread = toread - 1
ENDDO readchar
IF(PRESENT(iostat)) iostat = 0
RETURN
9999 CONTINUE ! EOR terminator read
IF(PRESENT(iostat)) iostat = ist
ENDSUBROUTINE get_d_tset_s

SUBROUTINE get_u_tset_s(unit, string, set, separator, 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
  type(VARYING_STRING), INTENT(OUT), OPTIONAL :: separator

```

```

! the actual separator character from set
! found as the input string terminator
! returned as zero length if termination
! by maxlen or EOR
INTEGER, INTENT(IN), OPTIONAL :: maxlen
! if present indicates the maximum
! number of characters that will be
! read from the file
INTEGER, INTENT(OUT), OPTIONAL :: iostat
! if present used to return the status
! of the data transfer
! if absent errors cause termination
! reads string from unit starting at next character in the file and
! terminating at the end of record, occurrence of a character in set,
! or after reading maxlen characters.
CHARACTER :: buffer ! characters must be read one at a time to detect
! first terminator character in set
INTEGER :: ist, toread, lenset
lenset = LEN(set)
IF(PRESENT(maxlen)) THEN
    toread = maxlen
ELSE
    toread = HUGE(1)
ENDIF
string = "" ! clears return string
IF(PRESENT(separator)) separator = "" ! clear the separator
readchar: DO ! repeatedly read buffer and add to string
    IF(toread <= 0) EXIT readchar ! maxlen reached
    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 GET of varying string on unit ", unit
            STOP
        ENDIF
    ENDIF
    ! check for occurrence of set character in buffer
    DO j = 1, lenset
        IF(buffer == set%chars(j)) THEN
            IF(PRESENT(separator)) separator = buffer
            EXIT readchar ! separator terminator found
        ENDIF
    ENDDO
    string = string // buffer
    toread = toread - 1
ENDDO readchar
IF(PRESENT(iostat)) iostat = 0
RETURN
9999 CONTINUE ! EOR terminator found
IF(PRESENT(iostat)) iostat = ist
ENDSUBROUTINE get_u_tset_s

SUBROUTINE get_d_tset_c(string, set, separator, maxlen, iostat)
type(VARYING_STRING), INTENT(OUT) :: string
! the string variable to be filled with
! characters read from the
! file connected to the default unit
CHARACTER(LEN=*), INTENT(IN) :: set
! the set of characters which if found in
! the input terminate the read
type(VARYING_STRING), INTENT(OUT), OPTIONAL :: separator
! the actual separator character from set
! found as the input string terminator
! returned as zero length if termination
! by maxlen or EOR
INTEGER, INTENT(IN), OPTIONAL :: maxlen
! if present indicates the maximum
! number of characters that will be
! read from the file

```

```

INTEGER, INTENT(OUT), OPTIONAL      :: iostat
                                     ! if present used to return the status
                                     ! of the data transfer
                                     ! if absent errors cause termination
! reads string from the default unit starting at next character in the file and
! terminating at the end of record, occurrence of a character in set,
! or after reading maxlen characters.
CHARACTER :: buffer ! characters must be read one at a time to detect
                   ! first terminator character in set
INTEGER    :: ist, toread, lenset
lenset = LEN(set)
IF(PRESENT(maxlen)) THEN
  toread = maxlen
ELSE
  toread = HUGE(1)
ENDIF
string = "" ! clears return string
IF(PRESENT(separator)) separator = "" ! clear separator
readchar: DO ! repeatedly read buffer and add to string
  IF(toread <= 0) EXIT readchar ! maxlen reached
  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 GET of varying string on default unit"
      STOP
    ENDIF
  ENDIF
  ! check for occurrence of set character in buffer
  DO j = 1, lenset
    IF(buffer == set(j:j)) THEN
      IF(PRESENT(separator)) separator = buffer
      EXIT readchar
    ENDIF
  ENDDO
  string = string // buffer
  toread = toread - 1
ENDDO readchar
IF(PRESENT(iostat)) iostat = 0
RETURN
9999 CONTINUE ! EOR terminator read
IF(PRESENT(iostat)) iostat = ist
ENDSUBROUTINE get_d_tset_c

SUBROUTINE get_u_tset_c(unit, string, set, separator, 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
  type(VARYING_STRING), INTENT(OUT), OPTIONAL :: separator
                           ! the actual separator character from set
                           ! found as the input string terminator
                           ! returned as zero length if termination
                           ! by maxlen or EOR
  INTEGER, INTENT(IN), OPTIONAL :: maxlen
                           ! if present indicates the maximum
                           ! number of characters that will be
                           ! read from the file
  INTEGER, INTENT(OUT), OPTIONAL :: iostat
                           ! if present used to return the status
                           ! of the data transfer
                           ! if absent errors cause termination
! reads string from unit starting at next character in the file and

```

```

! terminating at the end of record, occurrence of a character in set,
! or after reading maxlen characters.
CHARACTER :: buffer ! Characters must be read one at a time to detect
                    ! first terminator character in set
INTEGER    :: ist,toread,lenset
lenset = LEN(set)
IF(PRESENT(maxlen))THEN
    toread=maxlen
ELSE
    toread=HUGE(1)
ENDIF
string = "" ! clears return string
IF(PRESENT(separator)) separator="" ! clear separator
readchar:DO ! repeatedly read buffer and add to string
    IF(toread <= 0)EXIT readchar ! maxlen reached
    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 GET of varying string on unit ",unit
            STOP
        ENDIF
    ENDIF
    ! check for occurrence of set character in buffer
    DO j = 1,lenset
        IF(buffer == set(j:j))THEN
            IF(PRESENT(separator)) separator=buffer
            EXIT readchar ! separator terminator found
        ENDIF
    ENDDO
    string = string//buffer
    toread = toread - 1
ENDDO readchar
IF(PRESENT(iostat)) iostat = 0
RETURN
9999 CONTINUE ! EOR terminator read
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 PUT 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

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

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

```

ENDFUNCTION replace_cs

```

FUNCTION replace_cs_sf(string,start,finish,substring)
  type(VARYING_STRING)      :: replace_cs_sf
  CHARACTER(LEN=*) , INTENT(IN)  :: string
  INTEGER, INTENT(IN)         :: start, finish
  type(VARYING_STRING), INTENT(IN) :: substring
  ! calculates the result string by the following actions:
  ! inserts the substring into string beginning at position
  ! start replacing the following finish-start+1 characters of the string
  ! and enlarging or shrinking the string if necessary.
  ! If start is greater than LEN(string) substring is simply appended to string
  ! by concatenation. If start is less than 1, start = 1 is used
  ! If finish is greater than LEN(string), finish = LEN(string) is used
  ! If finish is less than start, substring is inserted before start
  CHARACTER, POINTER, DIMENSION(:) :: work
  INTEGER                          :: ip, is, if, nw, lsub, ls
  lsub = LEN(substring); ls = LEN(string)
  is = MAX(start, 1)
  ip = MIN(ls+1, is)
  if = MAX(ip-1, MIN(finish, ls))
  nw = lsub + ls - if + ip - 1
  ALLOCATE(work(1:nw))
  DO i=1, ip-1
    work(i) = string(i:i)
  ENDDO
  work(ip:ip+lsub-1) = substring%chars
  DO i=1, nw-ip-lsub+1
    work(i+ip+lsub-1) = string(if+i:if+i)
  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-i+1) = 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 occurrences of target in string and replaces these with
! substring. if back present with value true search is backward otherwise
! search is done forward. if every present with value true all occurrences
! of target in string are replaced, otherwise only the first found is
! replaced. if target is not found the result is the same as string.
LOGICAL :: dir_switch, rep_search
CHARACTER, DIMENSION(:), POINTER :: work, temp
INTEGER :: ls, lt, lsub, ipos, ipow
ls = LEN(string); lt = LEN(target); lsub = LEN(substring)
IF(lt==0)THEN
    WRITE(*,*) " Zero length target in REPLACE"
    STOP
ENDIF
ALLOCATE(work(1:ls)); work = string%chars
IF( PRESENT(back) )THEN
    dir_switch = back
ELSE
    dir_switch = .FALSE.
ENDIF
IF( PRESENT(every) )THEN
    rep_search = every
ELSE
    rep_search = .FALSE.
ENDIF
IF( dir_switch )THEN ! backwards search
    ipos = ls-lt+1
    DO
        IF( ipos < 1 )EXIT ! search past start of string
        ! test for occurrence of target in string at this position
        IF( ALL(string%chars(ipos:ipos+lt-1) == target%chars) )THEN
            ! match found allocate space for string with this occurrence of
            ! target replaced by substring
            ALLOCATE(temp(1:SIZE(work)+lsub-lt))
            ! copy work into temp replacing this occurrence of target by
            ! substring
            temp(1:ipos-1) = work(1:ipos-1)
            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
    END DO

```

```

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

```

```

        temp(ipos+1sub:) = work(ipos+1t:)
        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-1t+1
    ENDIF
    ipos=ipos-1
ENDDO
ELSE ! forward search
    ipos = 1; ipow = 1
    DO
        IF( ipos > 1s-1t+1 )EXIT ! search past end of string
        ! test for occurrence of target in string at this position
        IF( ALL(string%chars(ipos:ipos+1t-1) == target%chars) )THEN
            ! match found allocate space for string with this occurrence of
            ! target replaced by substring
            ALLOCATE(temp(1:SIZE(work)+1sub-1t))
            ! copy work into temp replacing this occurrence of target by
            ! substring
            temp(1:ipow-1) = work(1:ipow-1)
            DO i=1,1sub
                temp(i+ipow-1) = substring(i:i)
            ENDDO
            temp(ipow+1sub:) = work(ipow+1t:)
            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+1t-1; ipow = ipow+1sub-1
        ENDIF
        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 occurrences of target in string, and replaces these with
    ! substring. if back present with value true search is backward otherwise
    ! search is done forward. if every present with value true all occurrences
    ! of target in string are replaced, otherwise only the first found is
    ! replaced. if target is not found the result is the same as string.
    LOGICAL                        :: dir_switch, rep_search
    CHARACTER,DIMENSION(:),POINTER :: work,temp,tget
    INTEGER                        :: 1s,1t,1sub,ipos,ipow
    1s = LEN(string); 1t = LEN(target); 1sub = LEN(substring)
    IF(1t==0)THEN
        WRITE(*,*) "Zero length target in REPLACE"
        STOP
    ENDIF
    ALLOCATE(work(1:1s)); work = string%chars
    ALLOCATE(tget(1:1t))
    DO i=1,1t
        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 = 1s-1t+1
        DO

```

```

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

```

```

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

```

```

not found the result is the same as string.
      :: dir_switch, rep_search
ENTER :: work,temp
      :: ls,lt,ls,lt,ipos,ipow
(target); ls = LEN(substring)

target in REPLACE"

kwards search

search past start of string
of target in string at this position
t-1) == target )THEN
te space for string with this occurrence of
 substring
(work)+ls-lt))
p replacing this occurrence of target by

k(1:ipos-1)
1) = substring%chars
work(ipos+lt:)
new version of work point at the temp space
EXIT ! exit if only first replacement wanted
placement positions over the effected positions

```

```

        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+ls-1
    ENDIF
    ipos=ipos+1; ipow=ipow+1
ENDDO
ENDIF
replace_ccs%chars => work
ENDFUNCTION replace_ccs

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

```