Information technology – Programming languages – Fortran –

Part 2:
Varying Length Character Strings

[This page to be replaced by ISO CS.]

Foreword

[General part to be provided by ISO CS]

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 1539-1 : 1997, 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 ISO/IEC 1539-1 : 1997 is informally known as Fortran 95.

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 refers to a possible implementation, in Fortran 95, 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 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.

CONTENTS

ISO/IEC 1539-2: 1999(E) – Information technology – Programming languages – Fortran – Part 2: Varying Length Character Strings

1 General

1.1 Scope

This part of ISO/IEC 1539 defines facilities 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 specified by ISO/IEC 1539-1 and informally known as Fortran 95.

1.2 Changes from the previous version

This standard is a development from a previous version known as ISO/IEC 1539-2: 1994 that takes account of the improvements introduced in Fortran 95. The most significant improvements in Fortran 95 for the present standard were the introduction of pure and elemental procedures. Since pure and elemental functions can be used in specification expressions, their introduction in this standard enhances the usability of the standard for the end user. The ability to define many of the functions specified in this standard to be elemental improves the compatibility of these functions with similar intrinsic functions defined by the main standard.

The improvements in type initialization provided in Fortran 95 have also enabled the sample implementation referred to in Annex A to be written in such a way that significant leakage of memory is less likely to occur.

The following summarises the changes made to the facilities provided by the standard :–

The assignment, concatenation, and comparison operations are extended to describe elemental semantics.

CHAR is extended to describe pure semantics.

ADJUSTL, ADJUSTR, EXTRACT, IACHAR, ICHAR, INDEX, INSERT, LEN, LEN_TRIM, LGE, LGT, LLE, LLT, REMOVE, REPEAT, REPLACE, SCAN, SPLIT, TRIM, VAR_STR, and VERIFY are all extended to describe elemental semantics.

1.3 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 1539-1 : 1997 *Information technology – Programming Languages – Fortran.*

2 Overview

This part of ISO/IEC 1539 is an auxiliary standard to that defining Fortran 95 in that it defines additional facilities to those defined intrinsically in the primary language standard. A processor conforming to the Fortran 95 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 95 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 1539-1 : 1997.

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 1539-1 : 1997.

A module, written in standard conforming Fortran, is referenced 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 referenced

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.

3 Requirements

3.1 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
```

3.2 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.

3.3 Extended Meanings for Intrinsic Operators

The meanings for the intrinsic operators of:

assignment =

concatenation //

comparisons ==, /=, <, <=, >=, >

shall be extended to accept any combination of 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.

3.3.1 Assignment

An elemental assignment of the form

var = expr

shall be defined with the following type combinations:

`VARYING_STRING` and `VARYING_STRING`
`VARYING_STRING` and `CHARACTER`
`CHARACTER` and `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.

3.3.2 Concatenation

The elemental concatenation operation

`string_a // string_b`

shall be defined with the following type combinations:

`VARYING_STRING` and `VARYING_STRING`
`VARYING_STRING` and `CHARACTER`
`CHARACTER` and `VARYING_STRING`

The values of the operands are unchanged by the operation.

Result Characteristics. 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.

3.3.3 Comparisons

Elemental comparisons of the form

```
string_a == string_b
string_a /= string_b
string_a < string_b
string_a <= string_b
string_a > string_b
string_a >= string_b
```

shall be defined for operands with the following type combinations:

`VARYING_STRING` and `VARYING_STRING`
`VARYING_STRING` and `CHARACTER`
`CHARACTER` and `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 Characteristics. 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.

3.4 Extended Meanings for Generic Intrinsic Procedures

The generic intrinsic procedures ADJUSTL, ADJUSTR, CHAR, IACHAR, ICHAR, INDEX, LEN, LEN_TRIM, LGE, LGT, LLT, LLE, REPEAT, SCAN, TRIM, and VERIFY shall have their meanings extended to include the appropriate argument type combinations involving VARYING_STRING and CHARACTER. Detailed descriptions of the extensions are given in this section.

3.4.1 ADJUSTL (string)

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

Class. Elemental function.

Argument. `string` shall be of type VARYING_STRING.

Result Characteristics. 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.

3.4.2 ADJUSTR (string)

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

Class. Elemental function.

Argument. `string` shall be of type VARYING_STRING.

Result Characteristics. 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.

3.4.3 CHAR (string [, length])

Description. Converts a varying string value to default CHARACTER.

Class. Pure transformational function.

Arguments.

`string` shall be scalar and of type VARYING_STRING.

`length` (optional) shall be scalar and of type default INTEGER.

Result Characteristics. 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.

> **Note.** This function is elemental in Fortran 95, where it has the form `CHAR(i [,kind])`, with `i` of type integer.

3.4.4 IACHAR (c)

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

Class. Elemental function.

Argument. `c` shall be of type `VARYING_STRING` and of length exactly one.

Result Characteristics. 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))`.

3.4.5 ICHAR (c)

Description. Returns the position of a character in the processor defined collating sequence.

Class. Elemental function.

Argument. `c` shall be of type `VARYING_STRING` and of length exactly one.

Result Characteristics. 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))`.

3.4.6 INDEX (string, substring [, back])

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

Class. Elemental function.

Arguments.

`string` and `substring` shall be of one of the type combinations:

`VARYING_STRING` and `VARYING_STRING`
`VARYING_STRING` and `CHARACTER`
`CHARACTER` and `VARYING_STRING`

`back` (optional) shall be of type default `LOGICAL`.

Result Characteristics. 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`,

(where EXTRACT is defined in Section 3.7) or zero if there is no such value.

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.

3.4.7 LEN (string)

Description. Returns the length of a character string.

Class. Elemental function.

Argument. `string` shall be of type `VARYING_STRING`. The argument is unchanged by the procedure.

Result Characteristics. Of type default `INTEGER`.

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

Note. This function is not elemental for `string` of type `CHARACTER`.

3.4.8 LEN_TRIM (string)

Description. Returns the length of a string not counting any trailing blanks.

Class. Elemental function.

Argument. `string` shall be of type `VARYING_STRING`.

Result Characteristics. 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.

3.4.9 LGE (string_a, string_b)

Description. Compares the lexical ordering of two strings based on the ISO 646 : 1991 collating sequence.

Class. Elemental function.

Arguments.

`string_a` and `string_b` shall be of one of the type combinations:

`VARYING_STRING` and `VARYING_STRING`
`VARYING_STRING` and `CHARACTER`
`CHARACTER` and `VARYING_STRING`

Result Characteristics. Of type default `LOGICAL`.

Result Value. The result value is true if `string_a` is lexically greater than or equal to `string_b`, and is false otherwise. The collating sequence used to establish the ordering of characters 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.

3.4.10 LGT (string_a, string_b)

Description. Compares the lexical ordering of two strings based on the ISO 646 : 1991 collating sequence.

Class. Elemental function.

Arguments.

`string_a` and `string_b` shall be of one of the type combinations:

VARYING_STRING and VARYING_STRING
VARYING_STRING and CHARACTER
CHARACTER and VARYING_STRING

Result Characteristics. Of type default LOGICAL.

Result Value. The result value is true if string_a is lexically greater than string_b, and is false otherwise. The collating sequence used to establish the ordering of characters 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.

3.4.11 LLE (string_a, string_b)

Description. Compares the lexical ordering of two strings based on the ISO 646 : 1991 collating sequence.

Class. Elemental function.

Arguments.

string_a and string_b shall be of one of the type combinations:

VARYING_STRING and VARYING_STRING
VARYING_STRING and CHARACTER
CHARACTER and VARYING_STRING

Result Characteristics. Of type default LOGICAL.

Result Value. The result value is true if string_a is lexically less than or equal to string_b, and is false otherwise. The collating sequence used to establish the ordering of characters 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.

3.4.12 LLT (string_a, string_b)

Description. Compares the lexical ordering of two strings based on the ISO 646 : 1991 collating sequence.

Class. Elemental function.

Arguments.

string_a and string_b shall be of one of the type combinations:

VARYING_STRING and VARYING_STRING
VARYING_STRING and CHARACTER
CHARACTER and VARYING_STRING

Result Characteristics. Of type default LOGICAL.

Result Value. The result value is true if string_a is lexically less than string_b, and is false otherwise. The collating sequence used to establish the ordering of characters 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.

3.4.13 REPEAT (string, ncopies)

Description. Concatenates several copies of a string.

Class. Elemental function.

Arguments.

`string` – shall be of type `VARYING_STRING`,

`ncopies` – shall be of type default `INTEGER`.

Result Characteristics. 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 the value of `ncopies` is not positive, the result is of zero length.

Note. This function is not elemental for `string` of type `CHARACTER`.

3.4.14 SCAN (string, set [, back])

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

Class. Elemental function.

Arguments.

`string` and `set` shall be of one of the type combinations:

`VARYING_STRING` and `VARYING_STRING`
`VARYING_STRING` and `CHARACTER`
`CHARACTER` and `VARYING_STRING`

`back` (optional) shall be of type default `LOGICAL`.

Result Characteristics. 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.

3.4.15 TRIM (string)

Description. Removes trailing blanks from a string.

Class. Elemental function.

Argument. `string` shall be of type `VARYING_STRING`.

Result Characteristics. 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.

Note. This function is not elemental for `string` of type `CHARACTER`.

3.4.16 VERIFY (string, set [, back])

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

Class. Elemental function.

Arguments.

`string` and `set` shall be of one of the type combinations:

`VARYING_STRING` and `VARYING_STRING`
`VARYING_STRING` and `CHARACTER`
`CHARACTER` and `VARYING_STRING`

`back` (optional) shall be of type default `LOGICAL`.

Result Characteristics. 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.

3.5 Additional Generic Procedure for Type Conversion

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

3.5.1 VAR_STR (char)

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

Class. Elemental function.

Argument. `char` shall be of type default `CHARACTER` and may be of any length.

Result Characteristics. Of type `VARYING_STRING`.

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

3.6 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

3.6.1 GET (string [, maxlen] [, iostat]) or GET (unit, string [, maxlen] [, iostat]) or GET (string, set [, separator] [, maxlen] [, iostat]) or GET (unit, string, set [, separator] [, maxlen] [, iostat])

Description. Reads characters from an external file into a string.

Class. Subroutine.

Arguments.

`string` shall be scalar and of type `VARYING_STRING`. It is an INTENT(OUT) argument.

`maxlen` (optional) shall be scalar and of type default `INTEGER`. It is an INTENT(IN) argument.

`unit` shall be scalar and of type default `INTEGER`. It is an INTENT(IN) argument that specifies the input unit to be used. The unit shall be connected to a formatted file for sequential read access. If the argument `unit` is omitted, the default input unit is used.

`set` shall be scalar and either of type `VARYING_STRING` or of type `CHARACTER`. It is an INTENT(IN) argument.

`separator` (optional) shall be scalar and of type `VARYING_STRING`. It is an INTENT(OUT) argument.

`iostat` (optional) shall be scalar and of type default `INTEGER`. It is an INTENT(OUT) argument.

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 shall 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.

3.6.2 PUT (string [, iostat]) or PUT (unit, string [, iostat])

Description. Writes a string to an external file.

Class. Subroutine.

Arguments.

`string` shall be scalar and either of type `VARYING_STRING` or type `CHARACTER`. It is an INTENT(IN) argument.

`unit` shall be scalar and of type default `INTEGER`. It is an INTENT(IN) argument that specifies the output unit to be used. The unit shall be connected to a formatted file for sequential write access. If the argument `unit` is omitted, the default output unit is used.

`iostat` (optional) shall be scalar and of type default `INTEGER`. It is an INTENT(OUT) argument.

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.

3.6.3 PUT_LINE (string [, iostat]) or PUT_LINE (unit, string [, iostat])

Description. Writes a string to an external file and ends the record.

Class. Subroutine.

Arguments.

`string` shall be be scalar and either of type `VARYING_STRING` or type `CHARACTER`. It is an INTENT(IN) argument.

`unit` shall be be scalar and of type default `INTEGER`. It is an INTENT(IN) argument that specifies the output unit to be used. The unit shall be connected to a formatted file for sequential write access. If the argument `unit` is omitted, the default output unit is used.

`iostat` (optional) shall be scalar and of type default `INTEGER`. It is an INTENT(OUT) argument.

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.

3.7 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.

`EXTRACT` – extract a section from a string
`INSERT` – insert a substring into a string
`REMOVE` – remove a section of a string
`REPLACE` – replace a substring in a string

SPLIT – split a string into two at the occurrence of a separator

3.7.1 EXTRACT (string [, start] [, finish])

Description. Extracts a specified substring from a string.

Class. Elemental function.

Arguments.

string shall be either of type VARYING_STRING or type default CHARACTER

start (optional) shall be of type default INTEGER.

finish (optional) shall be of type default INTEGER.

Result Characteristics. 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.

3.7.2 INSERT (string, start, substring)

Description. Inserts a substring into a string at a specified position.

Class. Elemental function.

Arguments.

string shall be either type VARYING_STRING or type default CHARACTER.

start shall be type default INTEGER.

substring shall be either type VARYING_STRING or type default CHARACTER.

Result Characteristics. 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.

3.7.3 REMOVE (string [, start] [, finish])

Description. Removes a specified substring from a string.

Class. Elemental function.

Arguments.

string shall be either of type VARYING_STRING or type default CHARACTER
start (optional) shall be of type default INTEGER.
finish (optional) shall be of type default INTEGER.

Result Characteristics. 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.

3.7.4 REPLACE (string, start, substring) or REPLACE (string, start, finish, substring) or REPLACE (string, target, substring [,every] [,back])

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

Class. Elemental function.

Arguments.

string shall be either of type VARYING_STRING or type default CHARACTER.

start shall be of type default INTEGER.

finish shall be of type default INTEGER.

substring shall be either of type VARYING_STRING or type default CHARACTER.

target shall be either of type VARYING_STRING or type default CHARACTER. It shall not be of zero length.

every (optional) shall be of type default LOGICAL.

back (optional) shall be of type default LOGICAL.

Result Characteristics. 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.

3.7.5 SPLIT (string, word, set [, separator] [, back])

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

Class. Elemental subroutine.

Arguments.

`string` shall be of type `VARYING_STRING`. It is an INTENT(INOUT) argument.

`word` shall be of type `VARYING_STRING`. It is an INTENT(OUT) argument.

`set` shall be either of type `VARYING_STRING` or type default `CHARACTER`. It is an INTENT(IN) argument.

`separator` (optional) shall be of type `VARYING_STRING`. It is an INTENT(OUT) argument.

`back` (optional) shall be of type default `LOGICAL`. It is an INTENT(IN) argument.

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. Module ISO_VARYING_STRING

A sample implementation of the module ISO_VARYING_STRING is available from ftp.nag.co.uk/sc22wg5 and is written in Fortran 95, conformant with the language as specified in the standard ISO/IEC 1539-1 : 1997. It is intended to be a portable implementation of a module conformant with this part of ISO/IEC 1539 family of standards. 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.

Annex B. Two examples

This annex includes some examples illustrating the use of facilities conformant with 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 these example programs are a correct implementation of the stated problems using this part of ISO/IEC 1539 and in valid Fortran code, no guarantee is given or implied that this code will produce correct results, or even that it will execute on any particular processor.

B.1 Word count

The first example performs a ***word count***. Note, it is not claimed that this program is the best way to code this problem, nor even that it is a good way, merely that it is a way of solving this simple problem using the facilities defined by use of the facilities defined in this part of ISO/IEC 1539.

```
PROGRAM word_count
!------------------------------------------------------------------------------!
! Counts the number of "words" contained in a file. The words are assumed to  !
! be terminated by any one of:                                                 !
! space,comma,period,!,?, or the EoR                                           !
! The file may have records of any length and the file may contain any number !
! of records.                                                                  !
! The program prompts for the name of the file to be subject to a word count  !
! and the result is written to the default output unit                         !
!------------------------------------------------------------------------------!
USE ISO_VARYING_STRING
IMPLICIT NONE
type(VARYING_STRING) :: line,fname
INTEGER              :: ierr,nd,wcount=0
fname = "initial string"
```

```
WRITE(*,ADVANCE='NO',FMT='(A)') " Input name of file?"
CALL GET(STRING=fname) ! read the required filename from the default
                       ! input unit assumed to be the whole of the record read
OPEN(UNIT=10,FILE=CHAR(fname)) ! CHAR(fname) converts to the type
                               ! required by FILE= specifier
file_read: DO  ! until EoF reached
  CALL GET(10,line,IOSTAT=ierr)  ! read next line of file
  IF(ierr == -1 .OR. ierr > 0 )EXIT file_read
  word_scan: DO ! until end of line
    nd=SCAN(line," ,.!?")  ! scan to find end of word
    IF(nd == 0)THEN  ! EoR is end of word
      nd = LEN(line)
      EXIT word_scan
    ENDIF
    IF(nd > 1) wcount=wcount+1  ! at least one non-terminator character
                                ! in the word
    line = REMOVE(line,1,nd)  ! strips the counted word and its terminator
                              ! from the line reducing its length before
                              ! rescanning for the next word
  ENDDO word_scan
  IF(nd > 0) wcount=wcount+1
ENDDO file_read
IF(ierr < 0)THEN
  WRITE(*,*) "No. of words in file =",wcount
ELSEIF(ierr > 0)THEN
  WRITE(*,*) "Error in GET file in word_count, No. ",ierr
ENDIF
ENDPROGRAM word_count
```

B.2 Vocabulary list

A second and rather more realistic example is one which extends the above trivial example by producing a full ***vocabulary list*** along with frequency of occurrence for each different word. Again there is no claim that this is in anyway an optimal coding of this problem. It is merely an example that uses some of the facilities defined by this part of ISO/IEC 1539.

```
PROGRAM vocabulary_word_count
!------------------------------------------------------------------------------!
! Counts the number of "words" contained in a file. The words are assumed to  !
! be terminated by any one of:                                                 !
! space,comma,period,!,?, or the EoR                                           !
! The file may have records of any length and the file may contain any number !
! of records.                                                                  !
! The program prompts for the name of the file to be subject to a word count  !
! and the result is written to the default output unit                         !
! Also builds a list of the vocabulary found and the frequency of occurrence   !
! of each different word.                                                      !
!------------------------------------------------------------------------------!
USE ISO_VARYING_STRING
IMPLICIT NONE
```

```
type(VARYING_STRING) :: line,word,fname
INTEGER              :: ierr,nd,wcount=0
!-------------------------------------------------------------------------------!
! Vocabulary list and frequency count arrays. The size of these arrays will     !
! be extended dynamically in steps of 100 as the used vocabulary grows          !
!-------------------------------------------------------------------------------!
type(VARYING_STRING),ALLOCATABLE,DIMENSION(:) :: vocab
INTEGER,ALLOCATABLE,DIMENSION(:)              :: freq
INTEGER                                       :: list_size=200,list_top=0
INTEGER :: i   ! loop index
INTEGER :: record_count=1
!-------------------------------------------------------------------------------!
! Initialise the lists and determine the file to be processed                   !
!-------------------------------------------------------------------------------!
ALLOCATE(vocab(1:list_size),freq(1:list_size))
WRITE(*,ADVANCE='NO',FMT='(A)') " Input name of file?"
CALL GET(STRING=fname)  ! read the required filename from the default
                  ! input unit assumed to be the whole of the record read
OPEN(UNIT=1,FILE=CHAR(fname))  ! CHAR(fname) converts to the type
                               ! required by FILE= specifier
file_read: DO ! until EoF reached
  CALL GET(1,line,IOSTAT=ierr)  ! read next line of file
  IF(ierr == -1 .OR. ierr > 0)EXIT file_read
  WRITE(*,*) " record #",record_count," being processed"
  Record_count=record_count+1
  word_scan: DO ! until end of line
    nd=SCAN(line," ,.!?")  ! scan to find end of word
    IF(nd == 0)THEN  ! EoR is end of word
      nd = LEN(line)+1
      EXIT word_scan
    ENDIF
    IF(nd > 1)THEN  ! at least one non-terminator character in the word
       wcount=wcount+1
       word = EXTRACT(line,1,nd-1)
       CALL update_vocab_lists
    ENDIF
    line = REMOVE(line,1,nd)  ! strips the counted word and its terminator
                              ! from the line reducing its length before
                              ! rescanning for the next word
  ENDDO word_scan
  IF(nd > 1)THEN  ! at least one character in the word
     wcount=wcount+1
     word = EXTRACT(line,1,nd-1)
     CALL update_vocab_lists
   ENDIF
ENDDO file_read
IF(ierr < 0)THEN
  WRITE(*,*) "No. of words in file =",wcount
  WRITE(*,*) "There are ",list_top,"  distinct words"
  WRITE(*,*) "with the following frequencies of occurance"
  print_loop: DO i=1,list_top
```

```
    WRITE(*,FMT='(1X,I6,2X)',ADVANCE='NO') freq(i)
    CALL PUT_LINE(STRING=vocab(i))
  ENDDO print_loop
ELSEIF(ierr > 0)THEN
  WRITE(*,*) "Error in GET in vocabulary_word_count, No.",ierr
ENDIF

CONTAINS

SUBROUTINE extend_lists
!-----------------------------------------------------------------------------!
! Accesses the host variables:                                                !
!  type(VARYING_STRING),ALLOCATABLE,DIMENSION(:) :: vocab                     !
!  INTEGER,ALLOCATABLE,DIMENSION(:)              :: freq                      !
!  INTEGER                                       :: list_size                 !
! so as to extend the size of the lists preserving the existing vocabulary    !
! and frequency information in the new extended lists                         !
!-----------------------------------------------------------------------------!
type(VARYING_STRING),DIMENSION(list_size) :: vocab_swap
INTEGER,DIMENSION(list_size)              :: freq_swap
INTEGER,PARAMETER :: list_increment=100
INTEGER           :: new_list_size,alerr
vocab_swap = vocab  ! copy old list into temporary space
freq_swap =freq
new_list_size = list_size + list_increment
DEALLOCATE(vocab,freq)
ALLOCATE(vocab(1:new_list_size),freq(1:new_list_size),STAT=alerr)
IF(alerr /= 0)THEN
  WRITE(*,*) "Unable to extend vocabulary list"
  STOP
ENDIF
vocab(1:list_size) = vocab_swap   ! copy old list back into bottom
freq(1:list_size) = freq_swap     ! of new extended list
list_size = new_list_size
ENDSUBROUTINE extend_lists

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

```
    freq(i) = freq(i) + 1
    RETURN
  ENDIF
ENDDO list_search
IF(list_top == list_size)THEN
  CALL extend_lists
ENDIF
list_top = list_top + 1
vocab(list_top) = word
freq(list_top) = 1
ENDSUBROUTINE update_vocab_lists

ENDPROGRAM vocabulary_word_count
```