SPARK

SPARK95_IO - Input/Output for SPARK95 Programs

SPARK95_IO
Issue: 3.5
Status: Definitive
22 November 2005

Originator

SPARK Team

Approver

SPARK Team Line Manager

Contents

1 Introduction

The SPARK[1] 95 language has no predefined packages for Input-output, since the standard Ada input-output packages contain features not supported by SPARK.

However, the Examiner is supplied with a package `Spark_IO` which defines operations for file manipulation and input-output of the predefined types `Character`, `String`, `Integer` and `Float.` If required, facilities for input-output of new integer and floating point types, fixed point types and enumeration types may be provided by the user, based on procedures in `Spark_IO`, whose specification and body are supplied in machine-readable form with the SPARK Examiner.

This document describes the SPARK 95 variant of package `Spark_IO`.

The specification of the package `Spark_IO` obeys the rules of SPARK and can be used with other packages written in SPARK. Its subprograms, implemented as in the supplied version of the package body, will not raise unhandled exceptions.

As well as providing input-output facilities, `Spark_IO` also serves as a practical example of how to construct a SPARK interface to non-SPARK software, including use of the SPARK Examiner **hide** directive.

[1] Note: The SPARK programming language is not sponsored by or affiliated with SPARC International Inc. and is not based on SPARC™ architecture.

2 External Files and File Objects

Values input from the external environment of the program, or output to the environment, are considered to occupy *external* files. An external file can be anything external to the program that can produce a value or receive a value to be written. An external file is identified by a string - the *name.* A second string — the *form* — gives further system-dependent characteristics that may be associated with the file.

Input and output operations are expressed as operations on objects of some *file type,* rather than directly in terms of the external files. In the remainder of this chapter, the term *file* is always used to refer to a file object; the term *external file* is used otherwise. The values transferred for a given file must all be of one type.

The `Spark_IO` package declares three own variables `Inputs`, `Outputs` and `State`. `Inputs` represents the set of external input files used by a program and `Outputs` represents the set of external output files. `State` represents the internal essential state of the Spark_IO package. The file `Spark_IO.Standard_Input` is considered to be a member of `Inputs`, while `Spark_IO.Standard_Output` is considered to be a member of `Outputs`. A variable of type `Spark_IO.File_Type` is used to identify a particular file within the set of input or output files (just as an index is used to identify a particular element in an array); this variable does not represent the file itself.

The use of three own variables prevents unwanted coupling of outputs with inputs and complies with the guidelines in the INFORMED method for SPARK design.

Before performing any operation on an external file the file must have been opened (by `Open` if the file already exists or by `Create` if not). An open file has a current mode which is a value of the enumeration type

```
    type File_Mode is (In_File, Out_File, Append_File);
```

Before reading from a file (with one of the `Get` operations), the file must have been opened for input (mode `In_File`). Before writing to a file (with one of the `Put` operations), the file must have been opened for output (mode `Out_File` or `Append_File`).

The standard Ada input-output routines generate exceptions if error conditions are encountered, but exceptions are not allowed in SPARK. Instead, where appropriate a routine has a status parameter, whose returned value indicates whether an error condition has arisen. (If this status is not checked by a program, after every call of the routine, the SPARK Examiner will report the existence of data-flow anomalies - specifically the updating without subsequent reading of the status variable).

If a status variable indicates that an error has occurred, any other returned values are undefined, and the onus is on the programmer to organise appropriate recovery actions. For routines which do not return status information, the programmer should establish that their pre-conditions are always satisfied.

3 Text Input-Output

This section describes the facilities provided by the package `Spark_IO` for input and output in human-readable form. The package `Spark_IO` replaces the predefined Ada package `Ada.Text_IO`.

3.1 Types and Constants

Spark_IO declares a number of types and constants used throughout the package. These are as follows:

```
type File_Type is private;
type File_Mode is (In_File, Out_File, Append_File);
type File_status is (Ok, Status_Error, Mode_Error, Name_Error,
                     Use_Error, Device_Error, End_Error, Data_Error,
                     Layout_Error, Storage_Error, Program_Error);

subtype Number_Base is Integer range 2 .. 16;

Standard_Input  : constant File_Type;
Standard_Output : constant File_Type;
Null_File       : constant File_Type;
```

3.2 File Management

The procedures and functions described in this section provide for the control of external files.

```
procedure Create(File          :    out File_Type;
                 Name_Of_File  : in     String;
                 Form_Of_File  : in     String;
                 Status        :    out File_Status)
--# global  in out State;
--# derives State,
--#         File,
--#         Status from State, Name_Of_File, Form_Of_File;
```

If `File` does not identify a file within `Outputs` a new file is added to `Outputs` and `File` will be set to identify it. A new external file is created, with the given name and form. This external file is then associated with the file within `Outputs` identified by `File`. The external file is opened.

The current mode of the specified file is set to `Out_File` and the page length and line length are unbounded. The current column, the current line and the current page numbers are set to one.

A null string for Name_Of_File specifies an external file that is not accessible after the completion of the main program (a temporary file). A null string for Form_Of_File specifies the use of the default options of the implementation for the external file.

The Status parameter is set to Ok if no error condition is encountered. It is set to Status_Error if the specified file is already open, to Name_Error if the string given as Name does not allow the identification of an external file, or to Use_Error if, for the specified mode, the environment does not support creation of an external file with the given name (in the absence of Name_Error) and form.

Note that trying to use either Standard_Input or Standard_Output as the File parameter is not allowed. By the rules of Ada, a constant cannot be used as an “out” parameter.

```
procedure Open(File          :    out File_Type;
               Mode_Of_File  : in     File_Mode;
               Name_Of_File  : in     String;
               Form_Of_File  : in     String;
               Status        :    out File_Status)
--# global  in out State;
--# derives State,
--#         File,
--#         Status from State, Mode_Of_File, Name_Of_File,
--#                     Form_Of_File;
```

If File does not identify a file within Inputs or Outputs a new file is added to the appropriate set and File will be set to identify it. The external file, of the given name, is then associated with the file within Inputs or Outputs identified by File. The external file is opened.

The current mode of the specified file is set to the given access mode. If the current mode of the specified file is Out_File the page length and line length are unbounded. The current column, the current line and the current page numbers are set to one.

The Status parameter is set to Ok if no error condition is encountered. It is set to Status_Error if the specified file is already open, to Name_Error if the string given as Name does not allow the identification of an external file (in particular if no external file with the given name exists), or to Use_Error if, for the specified mode, the environment does not support opening of an external file with the given name (in the absence of Name_Error) and form.

Note that trying to use either Standard_Input or Standard_Output as the File parameter is not allowed. By the rules of Ada a constant cannot be used as an “out” parameter.

```
procedure Close(File   : in     File_Type;
                Status :    out File_Status)
--# global  in out State;
--# derives State,
--#         Status   from State, File;
```

The association between the file in Inputs or Outputs identified by File and its associated external file is severed. The specified file is closed.

If the file has current mode Out_File or Append_File, has the effect of calling New_Page, unless the current page is already terminated; then outputs a file terminator.

The Status parameter is set to Ok if no error condition is encountered. It is set to Status_Error if the specified file is not open. It is set to Use_Error if File is Standard_Input or Standard_Output.

```
procedure Delete(File   : in     File_Type;
                 Status :    out File_Status)
--# global  in out State;
--# derives State,
--#         Status from State, File;
```

The external file associated with the file in Inputs or Outputs identified by File is deleted. The specified file is closed and the external file ceases to exist.

The Status parameter is set to Ok if no error condition is encountered. It is set to Status_Error if the given file is not open or to Use_Error if deletion of the external file is not supported by the environment or if File is Standard_Input or Standard_Output.

```
procedure Reset(File         : in out File_Type;
                Mode_Of_File : in     File_Mode;
                Status       :    out File_Status);
--# derives File,
--#         Status from File, Mode_Of_File;
```

The file identified by File is reset so that reading from or writing to its elements can be restarted from the beginning of the file.

If the file has the current mode Out_File or Append_File, has the effect of calling New_Page, unless the current page is already terminated; then outputs a file terminator. If the new file mode is Out_File or Append_File, the page and line lengths are unbounded. The current column, line and page numbers are set to one.

The Status parameter is set to Ok if no error condition is encountered. It is set to Status_Error if the specified file is not open and to Use_Error if the environment does not support resetting for the external file or if the environment does not support resetting to the specified mode for the external file.

Note that trying to use either Standard_Input or Standard_Output as the File parameter is not allowed. By the rules of Ada a constant cannot be used as an “in out” parameter.

```
function Valid_File(File : File_Type) return Boolean;
```

This function checks that File is a valid identification for a file and returns the result of the check.

```
function Mode(File : File_Type) return File_Mode;
```

Returns the current mode of the file identified by `File`. If the specified file is not open the result is undefined.

```
procedure Name(File         : in     File_Type;
               Name_Of_File :    out String;
               Stop         :    out Natural);
--# derives Name_Of_File,
--#         Stop         from File;
```

If `File` is currently valid and open, copies into `Name_Of_File` a string which uniquely identifies the external file currently associated with the file identified by `File` (and may thus be used in an `Open` operation). If an environment allows alternative specifications of the name (for example, abbreviations), the string copied should correspond to a full specification of the name. If `Name_Of_File` is not big enough to hold the string, the string is truncated and `Stop` is set to `Name_Of_File 'LENGTH + 1`. Otherwise `Stop` is set to the length of the string copied into `Name_Of_File` .

If the specified file is not open or is invalid then the result is undefined.

```
procedure Form (File         : in     File_Type;
                Form_Of_File :    out String;
                Stop         :    out Natural);
--# derives Form_Of_File,
--#         Stop         from File;
```

If `File` is valid and open, copies into `Form_Of_File` the form string for the external file currently associated with the file in `Inputs` or `Outputs` identified by `File`. If an environment allows alternative specifications of the form (for example, abbreviations using default options), the string copied should correspond to a full specification (that is, it should indicate explicitly all options selected, including default options). If `Form_Of_File` is not big enough to hold the string, the string is truncated and `Stop` is set to `Form_Of_File'Length + 1`. Otherwise `Stop` is set to the length of the string copied into `Form_Of_File` .

If the specified file is invalid or not open then the result is undefined.

```
function Is_Open(File : File_Type) return Boolean;
--# global State;
```

Returns `True` if the file identified by `File` is open (that is, if it is associated with an external file), otherwise returns `False`.

3.3 Default Input and Output Files

Since SPARK does not allow overloading or default parameters, file parameters cannot be omitted from input-output operations which require them. Hence, `Spark_IO` does not support the concept of default input and output files and no routines are provided for their manipulation.

3.4 Specification of Line and Page Lengths

`Spark_IO` does not currently support the setting of line and page lengths. Therefore, files of mode `Out_File` or `Append_File` always have unbounded line and page lengths (that is, they have the conventional value zero). New lines and new pages are only started when explicitly called for.

3.5 Operations on Columns, Lines and Pages

The subprograms described in this section provide for explicit control of line and page structure. Currently `Spark_IO` does not support page operations on files of mode `In_File`.

```
procedure New_Line(File      : in   File_Type;
                   Spacing   : in   Positive);
--# global    in out Outputs;
--# derives   Outputs from Outputs, File, Spacing;
```

Operates on a file of mode `Out_File`.

For a `Spacing` of one: Outputs a line terminator and sets the current column number to one. Then increments the current line number by one, except in the case that the current line number is already greater than or equal to the maximum page length, for a bounded page length; in that case a page terminator is output, the current page number is incremented by one, and the current line number is set to one.

For a `Spacing` greater than one, the above actions are performed `Spacing` times.

No action is performed if the mode of the file identified by `File` is not `Out_File` or `File` is not a valid file identifier.

```
procedure Skip_Line(File      : in   File_Type;
                    Spacing   : in   Positive);
--# global    in out Inputs;
--# derives   Inputs from Inputs, File, Spacing;
```

Operates on a file of mode `In_File`.

For a `Spacing` of one: Reads and discards all characters until a line terminator has been read, and then sets the current column number to one. If the line terminator is not immediately followed by a page terminator, the current line number is incremented by one. Otherwise, if the line terminator is immediately followed by a page terminator, then the page terminator is skipped, the current page number is incremented by one, and the current line number is set to one.

For a `Spacing` greater than one, the above actions are performed `Spacing` times, or until a file terminator is reached.

No action is performed if the mode of the file identified by `File` is not `In_File`, an attempt is made to read a file terminator or `File` is not a valid file identifier.

```
procedure New_Page (File : in   File_Type);
--# global in out Outputs;
--# derives Outputs from Outputs, File;
```

Operates on a file of mode `Out_File`.

Outputs a line terminator if the current line is not terminated, or if the current page is empty (that is, the current column and line numbers are both equal to one). Then outputs a page terminator, which terminates the current page. Adds one to the current page number and sets the current column and line numbers to one.

No action is performed if the mode of the file identified by `File` is not `Out_File` or `File` is not a valid file identifier.

```
function End_Of_Line(File : File_Type) return Boolean;
--# global  Inputs;
```

Operates on a file of mode `In_File`.

Returns `True` if `File` is a valid identifier of an open file of mode `In_File` and a line terminator or a file terminator is next; Returns `False` if `File` is a valid identifier of an open file of mode `In_File` and a line terminator or a file terminator is not next; otherwise the result is undefined.

```
function End_Of_File(File : File_Type) return Boolean;
--# global Inputs;
```

Operates on a file of mode `In_File`.

Returns `True` if `File` is a valid identifier of an open file of mode `In_File` and a file terminator is next, or if the combination of a line and/or a page terminator followed by a file terminator is next; Returns `False` if `File` is a valid identifier of an open file of mode `In_File` and a line terminator is not next, or if the combination of a line and/or a page terminator followed by a file terminator is not next; otherwise the result is undefined.

```
procedure Set_In_File_Col(File   : in   File_Type;
                          Posn   : in   Positive);
--# global   in out Inputs;
--# derives  Inputs from Inputs, File, Posn;
--# pre Mode (File) = In_File;
```

Reads (and discards) individual characters, line terminator, and page terminators, until the next character to be read has a column number that equals the value specified by `Posn`; there is no effect if the current column number already equals this value. Each transfer of a character or terminator maintains the current column, line and page numbers in the same way as a `Get` procedure. (Short lines will be skipped until a line is reached that has a character at the specified column position.)   No action is performed if an attempt is made to read a file terminator.

```
procedure Set_Out_File_Col(File : in File_Type;
                           Posn : in Positive);
--# global in out Outputs;
--# derives Outputs from Outputs, File, Posn;
--# pre Mode (File) = Out_File or
--#     Mode (File) = Append_File;
```

If the value specified by `Posn` is greater than the current column number, outputs spaces, adding one to the current column number after each space, until the current column number equals the specified value. If the value specified by `Posn` is less than the current column number, has the effect of calling `New_Line` (with a spacing of one), then outputs (`Posn` - `1`) spaces, and sets the current column number to the specified value.

No action is performed if the value specified by `Posn` exceeds the line length when the line length is bounded (that is, when it does not have the conventional value zero).

```
function In_File_Col(File : File_Type) return Positive;
--# global Inputs;
--# pre Mode (File) = In_File;
```

If `File` is a valid identifier of an open file then returns the current column number; otherwise the result is undefined.

```
function Out_File_Col(File : File_Type) return Positive;
--# global Outputs;
--# pre Mode (File) = Out_File or
--#     Mode (File) = Append_File;
```

If `File` is a valid identifier of an open file then returns the current column number; otherwise the result is undefined.

```
function In_File_Line(File : File_Type) return Positive;
--# global Inputs;
--# pre Mode (File) = In_File;
```

If `File` is a valid identifier of an open file then returns the current line number; otherwise the result is undefined.

```
function Out_File_Line(File : File_Type) return Positive;
--# global Outputs;
--# pre Mode (File) = Out_File or
--#     Mode (File) = Append_File;
```

If `File` is a valid identifier of an open file then returns the current line number; otherwise the result is undefined.

3.6 Get and Put Procedures

To avoid overloading, a set of get and put procedures is provided (for example `Get_Char`, `Put_Char`, `Get_String`, `Put_String`). Features that are common to these procedures are described in this section.

Most of the `Get` and `Put` procedures operate on files and so they have a file parameter, written first. (The exceptions are the “get from string” and “put to string” procedures). Unlike `Ada.Text_IO`, in `Spark_IO` this file parameter cannot be omitted. The `Get` procedures operate on a file of mode `In_File` and the `Put` procedures operate on a file of mode `Out_File` or `Append_File`.

The `Get` and `Put` procedures maintain the current column, line and page numbers of the specified file, in the same way as `Ada.Text_IO`.

3.7 Input-Output of Characters and Strings

For an item of type `Character` or `String` the following procedures are provided.

```
procedure Get_Char(File : in     File_Type;
                   Item :    out Character);
--# global in out Inputs;
--# derives       Inputs,
--#               Item      from Inputs, File;
```

Operates on a file of mode `In_File`.

After skipping any line terminators and any page terminators, reads the next character from the specified input file and returns the value of this character in the out parameter `Item`.

If an attempt is made to skip a file terminator, no action is performed and the value of `Item` is undefined.

```
procedure Get_Char_Immediate (File   : in     File_Type;
                              Item   :    out Character;
                              Status :    out File_Status);
--# global in out Inputs;
--# derives Inputs,
--#         Item,
--#         Status from Inputs,
--#                     File;
```

Operates on a file of mode `In_File`. Only the variant of Get_Immediate that waits for a character to become available is supported.

On return Status is one of Ok, Mode_Error or End_Error. See ALRM A.10.7

Item is set to Character'First if Status /= Ok

```
procedure Put_Char(File   : in   File_Type;
                   Item   : in   Character);
--# global in out Outputs;
--# derives       Outputs   from Outputs, File, Item;
```

Operates on a file of mode `Out_File` or `Append_File`.

If the line length of the specified output file is bounded (that is, does not have the conventional value zero), and the current column number exceeds it, has the effect of calling `New_Line` with a spacing of one. Then, or otherwise, outputs the given character to the file.

```
procedure Get_String(File : in     File_Type;
                     Item :    out String;
                     Stop :    out Natural);
--# global in out Inputs;
--# derives   Inputs,
--#           Item,
--#           Stop       from Inputs, File;
```

Operates on a file of mode `In_File`.

Determines the length of the given string and attempts that number of `Get_Char` operations for successive characters of the string. If characters are read, returns in `Stop` the index value such that `Item(Stop)` is the last character replaced (the index of the first character replaced is `Item'FIRST`). If no characters are read returns in `Stop` an index value which is one less than `Item'FIRST`.

```
procedure Put_String(File : in   File_Type;
                     Item : in   String;
                     Stop : in   Natural);
--# global in out Outputs;
--# derives   Outputs   from Outputs, File, Item,
--#                          Stop;
```

Operates on a file of mode `Out_File` or `Append_File`.

If `Stop` is zero determines the length of the given string and attempts that number of `Put_Char` operations for successive characters of the string. If `Stop` is less than or equal to `Item'Last` then characters from `Item'First` up to and including `Stop` are output. If `Stop` is larger than `Item'Last` then all characters in `Item` are output, followed by spaces up to and including the width specified by `Stop`.

```
procedure Get_Line(File : in     File_Type;
                   Item :    out String;
                   Stop :    out Natural)
--# global in out Inputs;
--# derives   Inputs,
--#           Item,
--#           Stop      from Inputs, File;
```

Operates on a file of mode `In_File`.

Replaces successive characters of the specified string by successive characters read from the specified input file. Reading stops if the end of the line is met, in which case the procedure `Skip_Line` is then called (in effect) with a spacing of one; reading also stops if the end of the string is met. Characters not replaced are left undefined.

If characters are read, returns in `Last` the index value such that `Item(Last)` is the last character replaced (the index of the first character replaced is `Item'FIRST`). If no characters are read, returns in Last an index value that is one less than `Item'FIRST`. This value is also returned if an attempt is made to skip a file terminator.

```
procedure Put_Line(File : in    File_Type;
                   Item : in    String
                   Stop : in    Natural);
--# global in out Outputs;
--# derives   Outputs  from Outputs, File, Item, Stop;
```

Operates on a file of mode `Out_File` or `Append_File`.

Calls the procedure `Put_String` for the given string, and then the procedure `New_Line` with a spacing of one.

3.8 Input-Output for Integer Types

Since SPARK does not support generic packages, input-output routines are only provided for the predefined integer type `Integer`.

Values are output as decimal or based literals, without underline characters or exponent and preceded by a minus sign if negative. The format is specified by a non-negative field width parameter. Values of bases are of the Integer subtype `Number_base`,

```
subtype Number_base is Integer range 2 .. 16;
```

Since SPARK does not allow the specification of default parameters there is no default field width or base.

```
procedure Get_Integer(File  : in     File_Type;
                      Item  :    out Integer;
                      Width : in     Natural;
                      Read  :    out Boolean);
--# global in out Inputs;
--# derives   Inputs,
--#           Item,
--#           Read       from Inputs, File, Width;
```

Operates on a file of mode `In_File`.

If the value of the parameter Width is zero, skips any leading blanks, line terminators, or page terminators, then reads a plus or a minus sign if present, then reads according to the syntax of an integer literal (which may be a based literal). If a non-zero value of Width is supplied, then exactly Width characters are input, or the characters (possibly none) up to a line terminator, whichever comes first; any skipped leading blanks are included in the count.

If successful sets Read to True and returns, in the parameter Item, the Integer that corresponds to the sequence input. Otherwise sets Read to False and the value of Item is undefined.

```
procedure Put_Integer(File  : in    File_Type;
                      Item  : in    Integer;
                      Width : in    Natural;
                      Base  : in    Number_base);
--# global in out Outputs;
--# derives   Outputs   from Outputs, File, Item,
--#                            Width, Base;
```

Operates on a file of mode Out_File or Append_File.

Outputs the value of the parameter Item as an integer literal, with no underlines, no exponent, and no leading zeros (but a single zero for the value zero), and a preceding minus sign for a negative value.

If the resulting sequence of characters to be output has fewer than Width characters, then leading spaces are first output to make up the difference.

Uses the syntax for decimal literal if the parameter Base has the value ten; otherwise, uses the syntax for based literal, with any letters in upper case.

```
procedure Get_Int_From_String(Source   : in     String;
                              Item     :    out Integer;
                              Start_Pos: in     Positive;
                              Stop     :    out Natural);
--# derives   Item,
--#           Stop from Source, Start_Pos;
```

Reads an integer value from the beginning at Source(Start_Pos) from the given string, following the same rules as the Get_Integer procedure, but treating the end of the string as a file terminator. Returns, in the parameter Item, the Integer that corresponds to the sequence input. Returns in Stop the index value such that Source(Stop) is the last character read.

If the sequence input does not have the required syntax then Stop is one less than Start_Pos and the value of Item is undefined.

```
procedure Put_Int_To_String(Dest      : in out String;
                            Item      : in     Integer;
                            Start_Pos : in     Positive;
                            Base      : in     Number_Base);
```

```
    --# derives Dest from Dest, Item, Start_Pos, Base;
```

Outputs the value of the parameter `Item` to the given string such that the first digit (or sign) is at `Dest(Start_Pos)`, following the same rule as for output to a file, using `Dest'Last - Start_Pos + 1` as the value for `Width`.

3.9 Input-Output for Real Types

Since SPARK does not support generic packages, input-output routines are only provided for the predefined real type `Float`.

Values are output as decimal literals without underline characters. The format of each value consists of a `Fore` field, a decimal point, an `Aft` field, and (if a nonzero `Exp` parameter is supplied) the letter E and an `Exp` field.

Since SPARK does not allow the specification of default parameters there is no default `Fore`, `Aft` or `Exp`.

```
procedure Get_Float(File  : in     File_Type;
                    Item  :    out Float;
                    Width : in     Natural;
                    Read  :    out Boolean);
--# global in out Inputs;
--# derives   Inputs,
--#           Item,
--#           Read       from Inputs, File, Width;
```

Operates on a file of mode `In_File`.

If the value of the parameter `Width` is zero, skips any leading blanks, line terminators, or page terminators, then reads a plus or a minus sign if present, then reads according to the syntax of a float literal (which may be a based literal). If a non-zero value of `Width` is supplied, then exactly `Width` characters are input, or the characters (possibly none) up to a line terminator, whichever comes first; any skipped leading blanks are included in the count.

If successful sets `Read` to `True` and returns, in the parameter `Item`, the `Float` that corresponds to the sequence input. Otherwise sets `Read` to `False` and the value of `Item` is undefined.

```
procedure Put_Float(File  : in    File_Type;
                    Item  : in    Float;
                    Fore  : in    Natural;
                    Aft   : in    Natural;
                    Exp   : in    Natural);
--# global in out Outputs;
--# derives   Outputs   from Outputs, File, Item,
--#                          Fore, Aft, Exp;
```

Operates on a file of mode `Out_File` or `Append_File`.

Outputs the value of the parameter `Item` as a `Float` literal, with the format defined by `Fore`, `Aft` and `Exp`. If the value is negative, a minus sign is included in the integer part. If `Exp` has the value zero, then the integer part to be output has as many digits as are needed to represent the integer part of the value of `Item`, overriding `Fore` if necessary, or consists of the digit zero if the value of `Item` has no integer part.

```
procedure Get_Float_From_String
                           (Source    : in     String;
                            Item      :    out Float;
                            Start_Pos : in    Positive;
                            Stop      :    out Natural);
--# derives   Item,
--#           Stop   from Source, Start_Pos;
```

Reads a `Float` value starting at `Source(Start_Pos)` from the given string, following the same rules as the `Get_Float` procedure, but treating the end of the string as a file terminator. Returns, in the parameter Item the value that corresponds to the sequence input. Returns in `Stop` the index value such that `Source(Stop)` is the last character read.

If the sequence input does not have the required syntax then `Stop` is one less than `Start_Pos` and the value of `Item` is undefined.

```
procedure Put_Float_To_String(Dest      : in out String;
                              Item      : in     Float;
                              Start_Pos : in     Positive;
                              Aft       : in     Natural;
                              Exp       : in     Natural);
--# derives Dest from Dest, Item, Start_Pos, Aft, Exp;
```

Outputs the value of the parameter `Item` to the given string starting at `Dest(Start_Pos)`, following the same rule as for `Put_Float` to a file, using the `Dest'Last - Start_Pos + 1` as the value for `Fore`.

3.10 Input-Output for Enumeration Types

Spark_IO contains no predefined routines for the support of input-output for enumeration types.

3.11 Specification of the Package Spark_IO

```
with Ada.Text_IO;
package Spark_IO
  --# own State   : State_Type;
  --#     Inputs  : Inputs_Type;
  --#     Outputs : Outputs_Type;
  --# initializes State,
  --#             Inputs,
  --#             Outputs;
is
  --# type State_Type is abstract;
  --# type Inputs_Type is abstract;
  --# type Outputs_Type is abstract;

  type File_Type is private;
  type File_Mode is (In_File, Out_File, Append_File);
  type File_Status is (Ok, Status_Error, Mode_Error,
                       Name_Error, Use_Error,
                       Device_Error, End_Error,
                       Data_Error, Layout_Error,
                       Storage_Error, Program_Error);

  subtype Number_Base is Integer range 2 .. 16;

  Standard_Input  : constant File_Type;
  Standard_Output : constant File_Type;
  Null_File       : constant File_Type;



-- File Management

  procedure Create( File         :    out File_Type;
                    Name_Of_File : in     String;
                    Form_Of_File : in     String;
                    Status       :    out File_Status);
    --# global in out State;
    --# derives State,
    --#         File,
    --#         Status   from State, Name_Of_File, Form_Of_File;
```

```
procedure Open( File         :    out File_Type;
                Mode_Of_File : in     File_Mode;
                Name_Of_File : in     String;
                Form_Of_File : in     String;
                Status       :    out File_Status);
  --# global in out State;
  --# derives State,
  --#         File,
  --#         Status from State, Mode_Of_File, Name_Of_File,
  --#                     Form_Of_File;

procedure Close( File   : in     File_Type;
                 Status :    out File_Status);
  --# global in out State;
  --# derives State,
  --#         Status   from State, File;

procedure Delete( File   : in     File_Type;
                  Status :    out File_Status);
  --# global in out State;
  --# derives State,
  --#         Status   from State, File;

procedure Reset( File         : in out File_Type;
                 Mode_Of_File : in     File_Mode;
                 Status       :    out File_Status);
  --# derives File,
  --#         Status   from File, Mode_Of_File;

function Valid_File( File : File_Type) return Boolean;

function Mode( File : File_Type) return File_Mode;

procedure Name( File         : in     File_Type;
                Name_Of_File :    out String;
                Stop         :    out Natural);
  --# derives Name_Of_File,
  --#         Stop          from File;
  procedure Form( File         : in      File_Type;
                  Form_Of_File :    out String;
                  Stop         :    out Natural);
    --# derives Form_Of_File,
    --#         Stop             from File;

  function Is_Open( File : File_Type) return Boolean;
  --# global State;

-- Control of default input and output Files

  --
  -- Not supported in Spark_IO
  --

-- Specification of line and page lengths

  --
  -- Not supported in Spark_IO
  --


-- Column, Line and Page Control

  procedure New_Line( File    : in File_Type;
                      Spacing : in Positive);
    --# global in out Outputs;
    --# derives Outputs from Outputs, File, Spacing;

  procedure Skip_Line( File    : in File_Type;
                       Spacing : in Positive);
    --# global in out Inputs;
    --# derives Inputs from Inputs, File, Spacing;

  procedure New_Page( File : in File_Type);
    --# global in out Outputs;
    --# derives Outputs from Outputs, File;

  function End_Of_Line( File : File_Type) return Boolean;
    --# global Inputs;
  function End_Of_File( File : File_Type) return Boolean;
    --# global Inputs;

  procedure Set_In_File_Col( File : in File_Type;
                             Posn : in Positive);
    --# global in out Inputs;
    --# derives Inputs from Inputs, File, Posn;
    --# pre Mode (File) = In_File;

  procedure Set_Out_File_Col( File : in File_Type;
                              Posn : in Positive);
    --# global in out Outputs;
    --# derives Outputs from Outputs, File, Posn;
    --# pre Mode( File ) = Out_File or
    --#     Mode (File) = Append_File;

  function In_File_Col( File : File_Type) return Positive;
    --# global Inputs;
    --# pre Mode (File) = In_File;

  function Out_File_Col( File : File_Type) return Positive;
    --# global Outputs;
    --# pre Mode (File) = Out_File or
    --#     Mode (File) = Append_File;

  function In_File_Line( File : File_Type) return Positive;
    --# global Inputs;
    --# pre Mode (File) = In_File;

  function Out_File_Line( File : File_Type) return Positive;
    --# global Outputs;
    --# pre Mode (File) = Out_File or
    --#     Mode (File) = Append_File;

-- Character Input-Output

  procedure Get_Char( File : in      File_Type;
                      Item :     out Character);
    --# global in out  Inputs;
    --# derives Inputs,
    --#         Item     from Inputs, File;
  procedure Put_Char( File : in File_Type;
                      Item : in Character);
    --# global in out Outputs;
    --# derives Outputs from Outputs, File, Item;


-- String Input-Output

  procedure Get_String( File     : in     File_Type;
                        Item     :    out String;
                        Stop     :    out Natural);
    --# global in out Inputs;
    --# derives Inputs,
    --#         Item,
    --#         Stop         from Inputs, File;

  procedure Put_String( File     : in File_Type;
                        Item     : in String;
                        Stop     : in Natural);
    --# global in out Outputs;
    --# derives Outputs from Outputs, File, Item, Stop;

  procedure Get_Line( File     : in     File_Type;
                      Item     :    out String;
                      Stop     :    out Natural);
    --# global in out Inputs;
    --# derives Inputs,
    --#         Item,
    --#         Stop     from Inputs, File;

  procedure Put_Line( File     : in File_Type;
                      Item     : in String;
                      Stop     : in Natural);
    --# global in out Outputs;
    --# derives Outputs from Outputs, File, Item, Stop;


-- Integer Input-Output

  -- Spark_IO only supports input-output of
  -- the built-in Integer type Integer
  procedure Get_Integer( File  : in     File_Type;
                         Item  :    out Integer;
                         Width : in     Natural;
                         Read  :    out Boolean);
    --# global in out Inputs;
    --# derives Inputs,
    --#         Item,
    --#         Read     from Inputs, File, Width;

  procedure Put_Integer( File  : in File_Type;
                         Item  : in Integer;
                         Width : in Natural;
                         Base  : in Number_Base);
    --# global in out Outputs;
    --# derives Outputs from Outputs, File, Item, Width, Base;

  procedure Get_Int_From_String( Source    : in     String;
                                 Item      :    out Integer;
                                 Start_Pos : in     Positive;
                                 Stop      :    out Natural);
    --# derives Item,
    --#         Stop from Source, Start_Pos;

  procedure Put_Int_To_String( Dest      : in out String;
                               Item      : in     Integer;
                               Start_Pos : in     Positive;
                               Base      : in     Number_Base);
    --# derives Dest from Dest, Item, Start_Pos, Base;


-- Float Input-Output

  -- Spark_IO only supports input-output of
  -- the built-in real type Float
  procedure Get_Float( File  : in     File_Type;
                       Item  :    out Float;
                       Width : in     Natural;
                       Read  :    out Boolean);
    --# global in out Inputs;
    --# derives Inputs,
    --#         Item,
    --#         Read     from Inputs, File, Width;

  procedure Put_Float( File  : in File_Type;
                       Item  : in Float;
                       Fore  : in Natural;
                       Aft   : in Natural;
                       Exp   : in Natural);
    --# global in out Outputs;
    --# derives Outputs from Outputs, File, Item, Fore, Aft, Exp;

  procedure Get_Float_From_String( Source    : in     String;
                                   Item      :    out Float;
                                   Start_Pos : in     Positive;
                                   Stop      :    out Natural);
    --# derives Item,
    --#         Stop from Source, Start_Pos;

  procedure Put_Float_To_String( Dest      : in out String;
                                 Item      : in     Float;
                                 Start_Pos : in     Positive;
                                 Aft       : in     Natural;
                                 Exp       : in     Natural);
    --# derives Dest from Dest, Item, Start_Pos, Aft, Exp;

private
--# hide Spark_IO;

  type IO_TYPE   is (Stdin, Stdout, NamedFile);
  type File_PTR  is access Ada.Text_IO.File_Type;

  -- In addition to the fields listed here, we consider the
  -- FILE_PTR.all record to contain the name and mode of the
  -- file from the point of view of the annotations above.
  type File_Type is record
         File    : File_Ptr := null;
         IO_Sort : IO_Type  := NamedFile;
       end record;

  Standard_Input  : constant File_Type := File_Type'(null, StdIn);
  Standard_Output : constant File_Type := File_Type'(null, StdOut);
  Null_File       : constant File_Type := File_Type'(null,
                                                     NamedFile);
end Spark_IO;
```

4 Exceptions in Input-Output

The standard Ada input-output routines generate exceptions if error conditions are encountered, but exceptions are not allowed in SPARK. Instead, where appropriate a routine has a status parameter, whose returned value indicates whether an error condition has arisen. The status parameter is of the following type.

```
type File_Status is (Ok, Status_Error, Mode_Error,
                     Name_Error, Use_Error,
                     Device_Error, End_Error,
                     Data_Error, Layout_Error,
                     Storage_Error, Program_Error);
```

If a status variable indicates that an error has occurred, any other returned values are undefined, and the onus is on the programmer to organise appropriate recovery actions. For routines which do not return status information, the programmer should establish that their pre-conditions are always satisfied.

5 Example of Input-Output

```
package Inventory
--# own         Content;
--# initializes Content;
is

   Max_Size : constant Integer := 100;

   type Inventories is limited private;
   type Part_Numbers is range 1000 .. 9999;

   procedure Add(Part   : in     Part_Numbers;
                 Number : in     Positive;
                 Full   :    out Boolean);
   --# global  in out Content;
   --# derives Content from Part, Number, Content &
   --#         Full    from Part, Content;

   procedure Look_Up(Part   : in     Part_Numbers;
                     Number :    out Natural);
   --# global  in Content;
   --# derives Number from Part, Content;

private

   type Sizes is range 0 .. Max_Size;
   subtype Indices is Sizes range 1 .. Sizes'Last;
   type Items is
          record
             Part_Number : Part_Numbers;
             Amount      : Positive;
             Empty       : Boolean;
          end record;
   type  Inventories is array (Indices) of Items;
end Inventory;
```

```
with Spark_IO, Inventory;
--# inherit Spark_IO, Inventory;
--# main_program
procedure Dialogue
--# global  in out Spark_IO.Inputs
--#                Spark_IO.Outputs, Inventory.Content;
--# derives Spark_IO.Inputs,
--#         Inventory.Content from * &
--#         Spark_IO.Outputs from *, Spark_IO.Inputs;
is
   Number : Inventory.Part_Numbers;
   Amount : Natural;

   procedure Set_Up_Inventory
   --# global  in out Inventory.Content;
   --# derives Inventory.Content from Inventory.Content;
   is
      Unused : Boolean;
   begin
      Inventory.Add(6520, 20,  Unused);
      Inventory.Add(2718, 17,  Unused);
      Inventory.Add(6046, 43,  Unused);
      Inventory.Add(9214, 10,  Unused);
      Inventory.Add(4933, 28,  Unused);
      Inventory.Add(4179, 173, Unused);
      Inventory.Add(7294, 87,  Unused);
   end Set_Up_Inventory;

   procedure Enter_Part(Number : out Inventory.Part_Numbers)
   --# global  in out Spark_IO.Inputs,
   --#                Spark_IO.Outputs;
   --# derives Spark_IO.Inputs   from * &
   --#         Spark_IO.Outputs  from *, Spark_IO.Inputs &
   --#         Number            from Spark_IO.Inputs;
   is
      Number_Read : Integer;
      Ok : Boolean;
   begin
      loop
         Spark_IO.Put_String(Spark_IO.Standard_Output,
                             "Part number? ", 0);
         Spark_IO.Get_Integer(Spark_IO.Standard_Input,
                              Number_Read, 0,Ok);
         exit when Ok and then
           (Number_Read >=
               Integer(Inventory.Part_Numbers'FIRST) and
            Number_Read <=
               Integer(Inventory.Part_Numbers'Last));
         Spark_IO.Put_Line(Spark_IO.Standard_Output,
                   "Invalid part number, try again", 0);
         Spark_IO.New_Line(Spark_IO.Standard_Output, 1);
      end loop;
      Number := Inventory.Part_Numbers(Number_Read);
   end Enter_Part;
```

```
begin -- Dialogue
   Set_Up_Inventory;
   loop
      Enter_Part(Number);
      Inventory.Look_Up(Number, Amount);
      Spark_IO.Set_Col(Spark_IO.Standard_Output, 5);
      Spark_IO.Put_String(Spark_IO.Standard_Output,
                          "Part Number: ", 0);
      Spark_IO.Put_Integer(Spark_IO.Standard_Output,
                           Integer(Number),
                           0, 10);
      Spark_IO.Put_String(Spark_IO.Standard_Output,
                        " - Items available:", 0);
      Spark_IO.Set_Out_File_Col(Spark_IO.Standard_Output, 50);
      if Amount = 0 then
         Spark_IO.Put_Line(Spark_IO.Standard_Output,
                           " NONE", 0);
         Spark_IO.New_Line(Spark_IO.Standard_Output, 1);
      else
         Spark_IO.Put_Integer(Spark_IO.Standard_Output,
                              Amount, 5, 10);
         Spark_IO.New_Line(Spark_IO.Standard_Output, 2);
      end if;

      exit when False; -- syntactic exit point for analysis
                       -- purposes
   end loop;
end Dialogue;
```

Example of an interaction (characters typed by the user are italicized) :

```
Part number? 450
Invalid part number, try again
Part number? 3456
      Part Number: 3456 - Items available:         NONE
Part number? 9214
      Part Number: 9214 - Items available:           10
```

Document Control and References

File under

$CVSROOT/userdocs/SPARK95_IO.doc

Changes history

Issue 0.1 (15th January 1998) Initial draft based on S.P0468.73.25

Issue 1.0 (21st January 1998) After formal review

Issue 1.1 (13th July 2000) Updated for Release 5.0

Issue 2.0 (19th July 2000) Definitive issue following review

Issue 2.1 (18th Nov 2002) New issue updated and corrected to accompany new edition of the SPARK Book and Examiner release 6.3.

Issue 3.0 (18th Nov 2002) Definitive issue following review S.P0468.79.78.

Issue 3.1 (9 June 2003) Conversion to new document format

Issue 3.2 (22 July 2004) Added Get_Char_Immediate

Issue 3.3 (1st December 2004) Company name changed, no other changes made.

Issue 3.4 (5th January 2005) Definitive issue following review S.P0468.79.88.

Issue 3.5 (22nd November 2005) Line Manager change.

Changes forecast

None