A General File Interface for the Optimization Program Inverse

(FOR VERSION 3.11)

Igor Grešovnik

Ljubljana, 27 September, 2005

Contents:

7. A GENERAL FILE INTERFACE FOR PROGRAMME INVERSE

7.1 Introduction

The file interface enables the programme to exchange data with other programmes through text files. Functions of the interface are designed for searching for various items in files, reading various types of data from files and writing data to files in various formats.

Exploiting the functionality of the interface, the programme can work with any simulation programme which uses text files for defining the input data and writing the results. Any piece of data in the text files which are arranged according to some rules can be located, read or replaced by other data using the interface functions. Only some primitive programming skills and some knowledge about the format of the files are necessary to programme any data exchange between the shell and the simulation programme which can possibly be needed during the optimization procedure.

Thanks to the general file interface, a simulation programme and the shell does not have to be integrated to use them together for solving optimization and inverse problems. This is due to the fact that the only contact points between the programmes in the optimization scheme are updating the input data for the simulation according to the current values of the design parameters and reading the results of the simulation which are involved in the value of the objective function and its derivatives.

Nevertheless, the integration is still necessary in the cases where operations which are typically done by the shell can be performed efficiently only using the tools which are possessed by the simulation programme. If this is the case, the shell should have a direct access to specific functions of the simulation programme. In some cases this is possible only when the shell and the programme are properly integrated. The transformation of the input data according to the current values of parameters and the manipulation of the results to evaluate the values of the objective function and its derivatives are operations for which it is often beneficial to use the functionality of the simulation programme.

There is another concerning aspect of using the file interface: the speed. Typically the simulation programmes output large amounts of results. Writing large amounts of results to files and searching for the needed data in such files is time consuming and can be a bottleneck in the optimization procedure. A direct interface makes possible to avoid these operations since the data can be directly transferred between the memory locations

at the optimization and simulation part where it normally resides during the programme runtime. Besides, only the needed portions of the data can be transferred.

More sophisticated simulation programmes make possible to precisely specify which results are to be output to a file. In this case the additional time needed for data exchange through files is negligible as compared with the time needed for the simulation. With such programmes the general file interface can be used almost as efficiently as would be a direct interface.

7.2 Structure and Philosophy of the General File Interface

The file interpreter functions of the interface enable searching for specific data in the interface files, reading data from these files, writing data to these files, controlling and influencing the state of these files, and controlling the interface system. The expression evaluator's functions of the interface enable getting information about the interface system and the interface files.

The interpreter functions which perform searching, reading and controlling the state of the interface files are divided into two groups. The first group of functions perform these operations on the pre-defined file *infile*, and the other group of analogous functions perform analogous operations on arbitrary files defined during the programme runtime. Such division is introduced because of greater simplicity. The user should not have problems by remembering the names and syntax of both groups of functions because each function from one set has an analogous function with similar name and syntax in the other. Functions from the second group have a sub-string "*file*" instead of "*f*" in their names (e.g. **filefindstring** instead of **ffindstring**) and have an additional argument - the specification of the file on which they operate. This additional argument is always the first one in the function's argument block and the other arguments are identical.

Functions which read the data, search for different items or write data to files always operate from the current position of the file, so the starting point of the operation does not have to be specified. After the operation is completed, the current position of the file is set to the most logical point according to the kind of operation. For the functions which read or write data such point is after the last read or written byte. There are in general two kinds of functions which search for data items. Functions of the first kind set the current position after a successfully performed operation to the position of the first byte of the found item while the functions of the second kind set the position to the first byte after the found item. There are exceptions to this rule. Functions **ffindbrac** and **filefindbrac** which search for closed pairs of brackets set the current position to the first byte after the opening bracket if there are any characters between the opening and the

closing bracket, and to the first character after the closing bracket if it follows immediately the opening bracket. Such behaviour is logical because it is expected that the operations on the file which will follow a search for a closed pair of bracket will start inside the bracket pair anything is contained in the brackets, otherwise these operations will start after the brackets. An error code is also recorded if the found brackets contain nothing since this is often an unexpected situation.

The described arrangement simplifies working with the interface and enables the user to use as few intermediate variables as possible. However, in many cases it does not suffice for exchanging data through files. Sometimes we need to jump to arbitrary positions which are somehow connected to operations which were previously performed on a file. A special set of functions was created to enable this by remembering and setting the current position in a specific file. The expression evaluator's functions **getfpos** and **getfilepos** return the current position of a file. The file interpreter functions **fmarkpos** and **filemarkpos** assign the current position of a file to the expression evaluator's variable the name of which is specified in the argument block of the corresponding function. The file interpreter functions **fsetpos** and **filesetpos** set the current position of a file to the value specified in the argument block of the corresponding function. Sometimes the functions **fincreasepos** and **fileincreasepos** which increase or decrease the file position are more appropriate.

Another mechanism of remembering characteristic positions during the interfacing operations is available. Most of the functions for reading and writing allow the user to call them with additional arguments which specify the names of variables of the expression evaluator to which the characteristic positions of the corresponding operations. For example, in the argument blocks of the functions **ffindbrac**, **fskipbrac**, **filefindbrac** and **fieskipbrac** the user can optionally specify names of the expression evaluator's variables to which the position of the found opening and closing bracket are assigned, respectively. If only one name is specified, the position of opening bracket is assigned to the appropriate variable.

In general, in functions which search for different items, the user can additionally specify at most two names of expression evaluator's variables for marking characteristic positions. At functions which set the current position in the file to the beginning of the found item, the position of the found item is assigned to the first variable and the position of the first byte after the found item is set to the second variable. It is other way around at the functions which set the position after the found item. Only one instead of two variable names can also be specified. Functions for finding closed pairs of brackets have more specific behaviour (see above). At functions which search for characters only one additional variable name can be specified in their argument blocks (two would not make sense because a character has a constant length 1). The current position after a successfully performed operation is assigned to the corresponding variable.

If the appropriate operations are not performed successfully, unusual values are assigned to variables determined by additional arguments for marking characteristic positions of the operations. Values lesser than 1 which can not represent file positions are usually used for this purpose. Therefore, the user can also test if operations were

performed successfully through the values of these auxiliary variables. The same is not valid for functions which perform reading and writing.

At the functions which read data, two additional arguments for marking characteristic file positions of the appropriate operations are allowed in their argument blocks. The current position in the file before the beginning of the operation is assigned to the expression evaluator's variable corresponding to the first additional argument, and the position of the first byte of the last piece of data which was read is assigned to the variable corresponding to the second additional argument.

The functions for searching and reading have the corresponding analogous functions which perform the same operations in a limited range in files. Such analogous functions have similar names with a suffix "*to*". They also require one additional argument which specifies the end of their range of action (the beginning is determined by the current position in the file, as usual). This is the first argument at functions which operate on the pre-defined file *infile* and the second at functions which operate on arbitrary files (the file specification is the first argument at such functions).

The general file system has a special way of handling errors. Normally all file interpreter and expression evaluator's functions of the optimization notify the user about errors by writing error reports to the standard output and to the shell's output file. At the interface we have a specific situation that a failure of an operation may or may not mean an error, dependent on the situation. This is especially expressive at searching operations. Typically, when we search for specific data in a file which contains various data written according to some rules, a lot of failed search operations are performed. This does not indicate that some data is missing, but only gives us additional information which we need in order to locate the data.

Because only true erroneous situations should be reported as errors, there should be a possibility of distinguishing between expected and unexpected failures of operations. A mechanism which enable this is built into the interface system. Failures which do not necessarily mean errors are not reported automatically, but are registered so that they can be examined after the operation is completed. A decision can then be brought whether the failure means an error or not. An error report can then be written only in the case that the failure of the operation means an error, otherwise further actions are undertaken normally. Unexpected situations which are definitely errors are not treated in an usual manner so that error reports are immediately written to the standard output and the output file of the shell.

There are two mechanisms of recording failures which are not automatically reported as errors. File interpreter functions save their status of success which is a zero integer if the function was performed successfully or a non-zero integer if the function failed in any sense. The programme keeps only one code of success so that it is overwritten by the next executed interface function. The user can obtain the value of this code by the expression evaluator's function **fileoperror** called without arguments, and so check if the last operation was performed successfully. The file interpreter functions

printfileoperror and **fprintfileoperror** can be used to print a report about the success status of the last operation of the general file interface. These functions must also be called without arguments to write such report.

Beside keeping in memory the success code of the last performed operation of the general file interface, at all instances of failures the appropriate codes are pushed to a special stack. These codes are the same as the above mentioned success codes, the only difference is that code zero which identifies successful operation is not pushed on the stack. The user can retrieve information about these codes by the same functions as for getting information about the success codes of the last performed operation, only that in this case the error must be specified by the position on the stack of errors.

If the function **fileoperror** is called by argument 0, the number of errors on the stack is returned. If it is called by a positive number, the appropriate error is returned where the argument identifies the successive number of the error code on the stack. If the argument is negative, its absolute value identifies the successive number of the error on the stack counted backwards. If the value of the argument exceeds the number of errors on the stack, zero is returned.

The meaning of the argument at the functions **printfileoperror** and **fprintfileoperror** is similar, except that when the argument is zero, a report about all errors is printed, which is also the case if the argument is greater by absolute value than the number of errors.

The number of errors that can be stored on the error stack is limited. When the limit is exceeded, the first few errors are cleared. The user can make the programme report every occurance of this situation by the **reportfileoperrorexcess** function called by a non-zero argument.

7.3 Some General File Interpreter Functions of the Interface

7.3.1 Controlling the Interface System

7.3.1.1 ***reportfileoperrorexcess*** { <switch> }

Specifies wether or not writing reports when a few file operation errors are cleared due to the excess of maximum allowed number of errors on the error stack. If *switch* is non-zero, the programme will report when cuch situations will occur, otherwise it will not.

7.3.1.2 *fprintfileoperror* { <num> }

Prints a report about a specific error or errors which have occurred during the file operations, to the optimization shell's output file.

If the function is called without arguments, it prints the report about error status of the last file operation. Each file operation records its error status. This is a code of the last error which occurred during the operation and **is not automatically printed** to standard output or the shell's output file. These are normally used to indicate situations which don't necessarily mean unexpected behaviour, for example if a string which is searched for in a file is not found. In some situations this can be absolutely normal, therefore the appropriate function of the shell will not automatically report an error, but the user can still check if such situation has occurred, because in some occasions this can mean that something had gone wrong.

Code zero indicates that no error has occurred during this operation. An error status of the next file operation overwrites the error status, but the information about an error is not lost since non-zero error codes are pushed to a stack from which they can be retrieved and appropriate error messages can be printed. This is done by functions **printfileoperror** and **fprintfileoperror** when they are called with an argument (*num*) which specifies a number of error on the stack.

num can be specified in any standard way in which numbers are specified in argument blocks of commands (as a number, as an expression or as a variable in the system of the expression evaluator). If the value of *num* is zero, error message for all recorded error codes are printed to the output file. If it is a positive number not greater than the number of recorded error codes, a message for the *num*-th error is printed to the output file (if it is greater than the number of recorded errors, then error messages for all errors are printed). If the value of *num* is negative and its absolute value is not greater than the number of recorded error codes, then an error message for the *num*-th error code, counted from the end of the stack, is printed (again a report about all recorded error codes is printed if *-num* is greater than the number of recorded errors).

All error codes of the file operations can be cleared from the stack by the **clearfileoperrors** function. The user can so keep trace only about errors which occur from a specific point on.

It is good to know that the number of error codes which are remembered is limited. When the number of recorded error codes execs a specific pre-defined value, the first few errors are cleared from the stack.

7.3.1.3 *printfileoperror* { <num> }

Prints a report about a specific error or errors which have occurred during the file operations, to the standard output. Otherwise this function is identical to **fprintfileoperror**.

7.3.1.4 *dprintfileoperror* { <num> }

Prints a report about a specific error or errors which have occurred during the file operations, to the standard output and the programme's output file. Otherwise this function is identical to **fprintfileoperror**.

7.3.1.5 *fwritefileoperror* { <num> }

Prints an error string which corresponds to a specific error or errors which have occurred during the file operations, to the programme's output file. The meaning of optional argument *num* is the same as at function **fprintfileoperror**.

7.3.1.6 *writefileoperror* { <num> }

Prints an error string which corresponds to a specific error or errors which have occurred during the file operations, to the standard output. Otherwise this function is identical to **fwritefileoperror**.

7.3.1.7 *dwritefileoperror* { <num> }

Prints an error string which corresponds to a specific error or errors which have occurred during the file operations, to the standard output and the programme's output file.. Otherwise this function is identical to **fwritefileoperror**.

7.3.1.8 *clearfileoperrors* { }

This function clear all error codes from the stack of file operation error codes and also sets the error code of the last file operation to zero. It is used to record all file operation errors anew from a specific point.

7.3.1.9 *setfileopbuflength* { length }

Sets the size of the buffer which is used ad searching operations to *length*. The default buffer length is 200 bytes and can be changed at any time. It is ideal if the buffer length is of the same magnitude of order than the expected search length since in this case only one or a few reading operations will be performed to find what is searched for. This is good because accessing input-output devices is usually time consumable. On the other hand, it is not good if the buffer is much larger than the expected search length because in this case a lot of necessary data will be read from the file which is also time consuming. If the expected search length is extremely large, i.e. takes a significant portion of memory available on the system, the buffer length must be smaller than recommended above because in the opposite case we can have problems with insufficient memory resources.

7.4 File Interpreter Functions of the Interface which Operate on the Pre-defined File "infile"

7.4.1 Controlling the State of the File

7.4.1.1 *fprintfpos* { }

Prints a report about the current position in the file *infile* to the shell's output file.

7.4.1.2 *printfpos* { }

Prints a report about the current position in the file *infile* to the standard output.

7.4.1.3 *dprintfpos* { }

Prints a report about the current position in the file *infile* to the standard output and the programme's output fie.

7.4.1.4 *fprintfpart* { pos1 <pos2> }

Prints a part of the file *infile* to the shell's output file. If *pos2* is specified, the part of the file from position *pos1* to (including) *pos2* is printed. If only *pos1* is specified, *pos1* bytes from the current position on (including the current position) of the file is printed. It prints the file part in the form of a report where it also prints which part of the file is printed.

If both *pos1* and *pos2* are specified, they can be zero, and *pos2* can be greater than the length of the file. In this case, if *pos1* is zero, it is changed to 1, if *pos2* is zero, it is changed to the length of the file, and if *pos2* is greater than the length of the file, it is also changed to the length of the file before the operation is performed. If only *pos1* is specified, it must be greater than zero. If in this case *pos1* bytes from the current position would exceed the length of the file, *pos1* is reduced before the operation so that the length of the file is matched.

An error is recorded if *pos1* or *pos2* are invalid. If the file variable is not defined, an error report is written to the standard output and to the programme's output file. If it is not connected to a physical file, an appropriate error code is recorded.

7.4.1.5 *printfpart* { pos1 <pos2> }

Does the same as **fprintfpart**, except that it prints to standard output.

7.4.1.6 *dprintfpart* { pos1 <pos2> }

Does the same as **fprintfpart**, except that it prints to the standard output and to the programme's output file.

7.4.1.7 fwritefpart { pos1 <pos2> }

Prints a part of the file *infile* to the shell's output file. If *pos2* is specified, the part of the file from position *pos1* to (including) *pos2* is printed. If only *pos1* is specified, *pos1* bytes from the current position on (including the current position) of the file is printed. Only the file part is printed without any explanation or newlines, so this function is useful if the user wants to use the contents of a file as a part of the output which he generates on his own way.

If both *pos1* and *pos2* are specified, they can be zero, and *pos2* can be greater than the length of the file. In this case, if *pos1* is zero, it is changed to 1, if *pos2* is zero, it is changed to the length of the file, and if *pos2* is greater than the length of the file, it is also changed to the length of the file before the operation is performed. If only *pos1* is specified, it must be greater than zero. If in this case *pos1* bytes from the current position would exceed the length of the file, *pos1* is reduced before the operation so that the length of the file is matched.

An error is recorded if *pos1* or *pos2* are invalid. If the file is not defined, an error report is written to the standard output and to the programme's output file. If it is not connected to a physical file, an appropriate error code is recorded.

7.4.1.8 writefpart { pos1 <pos2> }

Does the same as **fwritefpart**, except that it prints to standard output.

7.4.1.9 dwritefpart { pos1 <pos2> }

Does the same as **fwritefpart**, except that it prints to standard output and the programme's output file.

7.4.1.10 *fsetpos* { pos }

Sets the current position in the file *infile* to the value of *pos*. *pos* should be a non-negative number not greater than the file size. If it is not an integer, it is rounded to the nearest integer. If it is zero, it is changed to the length of the file plus 1 before the operation is performed.

If it *pos* is greater than the length of the file, an error is recorded and the position is set to the end of the file (file length plus 1). If rounded *pos* is less than zero, an error is recorded and nothing else happens. If the file is not defined, an error report is written to the standard output and to the programme's output file. If it is not connected to a physical file, an appropriate error code is recorded.

7.4.1.11 fincreasepos { inc }

Increases the current position in the file *infile* by *inc*. *inc* is rounded to the nearest integer. It can be negative (in this case the current position is decreased) or zero.

If the new position would be less than 1, an error code is recorded and the current position is set to 1. If the new position would be greater than the file length, an error code is recorded and the current position is set to the end of the file (the length of the file plus 1).

If the file is not defined, an error report is written to the standard output and to the programme's output file. If it is not connected to a physical file, an appropriate error code is recorded.

7.4.1.12 *fmarkpos* { pos }

Marks the current position in the file *infile*. The position is assigned to the expression evaluator's variable named *pos*. If the position can not be determined, an integer value less than 1 is assigned to the variable.

If the file is not defined, an error report is written to the standard output and to the programme's output file. An error is not reported or recorded if the file is not connected to a physical file. In this case the marked position is less than one.

7.4.1.13 *fflush* { }

Empties the file buffer of the pre-defined file *infile* and writes the unsaved data on the disk.

This function is used only in special ocasions, for example when we debug the command file and would like to check the effect of file operations directly by editing a file. by executing the fflush command we make sure that all performed file operations take effect, since files are buffered and many operations only take immediate effect on the buffer, not on the file.

7.4.2 Searching for the Data

7.4.2.1 *ffindstring* { string <pos after> }

Searches for the string *string* in the file *infile* from the current position on. If the string is found, it sets the current position of this file to the first byte of the found string. *pos* and *after* are optional arguments which specify the names of expression evaluator's variables into which additional information is stored. The function assigns the position of the string to the variable named *pos* and the position of the first byte after the string to the variable named *after*.

If the file is not defined, an error report is written to the standard output and to the programme's output file. If it is not connected to a physical file or if the string is not found, an appropriate error code is recorded.

7.4.2.2 *ffindstringto* {to string <pos after> }

Like **ffindstring**, only that the operation is performed on a limited region of the file. Argument *to* specifies the upper limit of the action range.

7.4.2.3 fskipstring { string <after pos> }

Searches for the string *string* in the file *infile* from the current position on. If the string is found, it sets the current position of this file to the first byte after the found string. Optional argument *after* specifies the name of the expression evaluator's variable to which the function assigns this position, and *pos* specifies the variable to which this function assigns the position of the found string (i.e. the first byte if this string if the file).

If the file is not defined, an error report is written to the standard output and to the programme's output file. If it is not connected to a physical file or if the string is not found, an appropriate error code is recorded.

7.4.2.4 fskipstringto { to string <after pos> }

Like **fskipstring**, only that the operation is performed on a limited region of the file. Argument *to* specifies the upper limit of the action range.

7.4.2.5 *fmultfindstring* { {string1 string2 string3 string4 ...} <which pos after> }

Searches for the first occurrence of any of the strings *string1*, *string2*, *string3*, etc. in the file *infile* from the current position on. If it finds any of these strings, it sets the current position to the position of the found string. Strings must be in curly brackets.

If argument *which* is given, the function assigns the ordinary number of the found string to the expression evaluator variable named *which*. If argument *pos* is given, the function assigns the position of the found string to the appropriate expression evaluator variable, and if *after* is also given, the function assign the position of the first byte after the found string to the variable with such name.

If the file is not defined, an error report is written to the standard output and to the programme's output file. If it is not connected to a physical file or if the string is not found, an appropriate error code is recorded.

7.4.2.6 *fmultfindstringto* { to {string1 string2 string3 string4 ...} <which pos after> }

Like **fmultfindstring**, only that the operation is performed on a limited region of the file. Argument *to* specifies the upper limit of the action range.

7.4.2.7 *fmultskipstring* { {string1 string2 string3 string4 ...} <which after pos> }

Searches for the first occurrence of any of the strings *string1*, *string2*, *string3*, etc. in the file *infile* from the current position on. If it finds any of these strings, it sets the current position to the position of the first byte after the found string. Strings must be in curly brackets.

If argument *which* is given, the function assigns the ordinary number of the found string to the expression evaluator variable named *which*. If argument *after* is given, the function assign the position of the first byte after the found string to the expression evaluator's variable named *after*, and if *pos* is also specified, the function assign the position of the found string to the variable with such name.

If the file is not defined, an error report is written to the standard output and to the programme's output file. If it is not connected to a physical file or if the string is not found, an appropriate error code is recorded.

7.4.2.8 *fmultskipstringto* { to {string1 string2 string3 string4 ...} <which after pos> }

Like **fmultskipstring**, only that the operation is performed on a limited region of the file. Argument *to* specifies the upper limit of the action range.

7.4.2.9 *ffindcharacter* { charstring <pos> }

Searches in the file *infile* from the current position for the first occurrence of a character included in the string *charstring*. If the search is successful then it sets the current position in the file to the position of the found character.

Optional argument *pos* specifies a name of the expression evaluator's variable to which the position of the found character is assigned. If the search is not successful then a number less than 1 is assigned to that variable.

If the file is not defined, an error report is written to the standard output and to the programme's output file. If it is not connected to a physical file or if the character is not found, an appropriate error code is recorded.

7.4.2.10 *ffindcharacterto* { to charstring <pos> }

Like **ffindcharacter**, only that the operation is performed on a limited region of the file. Argument *to* specifies the upper limit of the action range.

7.4.2.11 *fskipcharacters* { charstring <pos> }

Searches in the file *infile* from the current position for the first occurrence of a character not included in the string *charstring*. If the search is successful then it sets the current position in the file to the position of the found character.

Optional argument *pos* specifies a name of the expression evaluator's variable to which the position of the found character is assigned. If the search is not successful then a number less than 1 is assigned to that variable.

If the file is not defined, an error report is written to the standard output and to the programme's output file. If it is not connected to a physical file or if the character is not found, an appropriate error code is recorded.

7.4.2.12 *fskipcharactersto* { to charstring <pos> }

Like **fskipcharacters**, only that the operation is performed on a limited region of the file. Argument *to* specifies the upper limit of the action range.

7.4.2.13 *ffindblank* { <pos> }

Searches in the file *infile* from the current position for the first occurrence of a blank character, i.e. space, tab, newline, carriage return or null character. If the search is successful then it sets the current position in the file to the position of the found character.

Optional argument *pos* specifies a name of the expression evaluator's variable to which the position of the found character is assigned. If the search is not successful then a number less than 1 is assigned to that variable.

If the file is not defined, an error report is written to the standard output and to the programme's output file. If it is not connected to a physical file or if the character is not found, an appropriate error code is recorded.

7.4.2.14 *ffindblankto* { to <pos> }

Like **ffindblank**, only that the operation is performed on a limited region of the file. Argument *to* specifies the upper limit of the action range.

7.4.2.15 *fskipblanks* { <pos> }

Searches in the file *infile* from the current position for the first occurrence of a non-blank character (blank characters are the space, tab, newline, carriage return and null character). If the search is successful then it sets the current position in the file to the position of the found character.

Optional argument *pos* specifies a name of the expression evaluator's variable to which the position of the found character is assigned. If the search is not successful then a number less than 1 is assigned to that variable.

If the file is not defined, an error report is written to the standard output and to the programme's output file. If it is not connected to a physical file or if the character is not found, an appropriate error code is recorded.

7.4.2.16 *fskipblanksto* { to <pos> }

Like **fskipblanks**, only that the operation is performed on a limited region of the file. Argument *to* specifies the upper limit of the action range.

7.4.2.17 *fnextline* { <pos> }

Sets the current position in the file *infile* to the beginning of the next line from the current position in this file. If such position is found, it sets the current position in this file to it.

Optional argument *pos* specifies a name of the expression evaluator's variable to which the position of the found position is assigned. If the search is not successful then a number less than 1 is assigned to that variable.

If the file is not defined, an error report is written to the standard output and to the programme's output file. If it is not connected to a physical file or if the next line can not be located (e.g. when the current position is in the last line of the file), an appropriate error code is recorded.

7.4.2.18 *ffindbrac* { bracstr <pos1 pos2> }

Searches in the file *infile* from the current position for the first occurrence of a closed bracket specified by *bracstr*. The first character of *bracstr* must be the character used for the opening bracket and the second character of the string must be the character used for the closing bracket. These characters may not be the same. Sequences which represent special characters can be used in this string. If the search is successful then the function sets the current position in the file to the position of the first character inside the bracket if the bracket contain any characters, otherwise it sets the current position to the position of the first character after the bracket.

Optional arguments *pos1* and *pos2* specify names of the expression evaluator's variables to which the position of the found opening and closing bracket is assigned. If the search is not successful then a number less than 1 is assigned to those variables.

If the file is not defined, an error report is written to the standard output and to the programme's output file. If it is not connected to a physical file or if a bracket is not found, an appropriate error code is recorded. An error code is also recorded if a bracket is found but does not contain any characters, i.e. the closing bracket immediately follow the opening bracket.

7.4.2.19 *ffindbracto* { to bracstr <pos1 pos2> }

Like **ffindbrac**, only that the operation is performed on a limited region of the file. Argument *to* specifies the upper limit of the action range.

7.4.2.20 *fskipbrac* { bracstr <pos1 pos2> }

Searches in the file *infile* from the current position for the first occurrence of a closed bracket specified by *bracstr*. The first character of *bracstr* must be the character used for the opening bracket and the second character of the string must be the character used for the closing bracket. These characters may not be the same. Sequences which represent special characters can be used in this string. If the search is successful then the

function sets the current position in the to the position of the first character after the bracket.

Optional arguments *pos1* and *pos2* specify names of the expression evaluator's variables to which the position of the found opening and closing bracket is assigned. If the search is not successful then a number less than 1 is assigned to those variables.

If the file is not defined, an error report is written to the standard output and to the programme's output file. If it is not connected to a physical file or if a bracket is not found, an appropriate error code is recorded.

7.4.2.21 *fskipbracto* { to bracstr <pos1 pos2> }

Like **fskipbrac**, only that the operation is performed on a limited region of the file. Argument *to* specifies the upper limit of the action range.

7.4.2.22 *ffindnumber* { < start next val > }

Searches in the file *infile* from the current position for the first occurrence of a string which can represent a number. The number can be written in any standard format in which numbers are written in text files. If a number is found, the current position in the file is set to the position of the first character of the string which represent the number.

Optional arguments *start*, *next* and *val* specify names of the expression evaluator's variables to which the position of the found number, the position of the first character after the number, and the value of the number are assigned, respectively. If the search is not successful then integer numbers less than 1 are assigned to *start* and *next*.

If the file is not defined, an error report is written to the standard output and to the programme's output file. If it is not connected to a physical file or if a number is not found, an appropriate error code is recorded.

7.4.2.23 *ffindnumberto* { to < start next val > }

Like **ffindnumber**, only that the operation is performed on a limited region of the file. Argument *to* specifies the upper limit of the action range.

7.4.2.24 *fskipnumber* { < next start val > }

Searches in the file *infile* from the current position for the first occurrence of a string which can represent a number. The number can be written in any standard format in which numbers are written in text files. If a number is found, the current position in the file is set to the position of the first character after the string which represent the number.

Optional arguments *next*, *start* and *val* specify names of the expression evaluator's variables to which the position of the first character after the number, the position of the found number, and the value of the number are assigned, respectively. If the search is not successful then integer numbers less than 1 are assigned to *next* and *start*.

If the file is not defined, an error report is written to the standard output and to the programme's output file. If it is not connected to a physical file or if a number is not found, an appropriate error code is recorded.

7.4.2.25 *fskipnumberto* { to < next start val > }

Like **fskipnumber**, only that the operation is performed on a limited region of the file. Argument *to* specifies the upper limit of the action range.

7.4.3 Reading the Data

7.4.3.1 *freadnumber* { varname <start next> }

Reads a number from the file *infile* and assigns its value to the expression evaluator's variable named *varname*. Reading starts at the current position in the file. A number must reside at that position, otherwise the operation fails (blank characters are allowed between the current position and a number). If the operation is successful, the current position in the file is set to the position of the first character after the read data.

An optional argument *start* specifies a name of the expression evaluator's variable to which the current position in the file before the operation is assigned, and an optional argument *next* specifies a name of the expression evaluator's variable to which the position of the first character after the read data is assigned. In the case that no data is read, the position before the operation is also assigned to the variable named *next*. These values are assigned to the expression evaluator's variables named *start* and *next* regardless of the success of the operation, therefore the success can not be examined through the values of these variables, but only through registered errors.

If the file is not defined, an error report is written to the standard output and to the programme's output file. If it is not connected to a physical file or if the number can not be read, an appropriate error code is recorded.

7.4.3.2 *freadnumberto* { to varname <start next> }

Like **freadnumber**, only that the operation is performed on a limited region of the file. Argument *to* specifies the upper limit of the action range.

7.4.3.3 *freadscalar* { scalspec <start next> }

Reads a number from the file *infile* and assigns it to a programme's scalar variable. *scalspec* is a specification of the variable to which the number is assigned. It must be given in a standard form consisting of the variable name and optionally the index table in square brackets. Reading starts at the current position in the file. A number must reside at that position, otherwise the operation fails (blank characters are allowed between the current position and a number). If the operation is successful, the current position in the file is set to the position of the first character after the read data.

An optional argument *start* specifies a name of the expression evaluator's variable to which the current position in the file before the operation is assigned, and an optional argument *next* specifies a name of the expression evaluator's variable to which the position of the first character after the read data is assigned. In the case that no data is

read, the position before the operation is also assigned to the variable named *next*. These values are assigned to the expression evaluator's variables named *start* and *next* regardless of the success of the operation, therefore the success can not be examined through the values of these variables, but only through registered errors.

If the file is not defined, an error report is written to the standard output and to the programme's output file. If it is not connected to a physical file or if the scalar can not be read, an appropriate error code is recorded.

7.4.3.4 ***freadscalarto*** { to scalspec <start next> }

Like **freadscalar**, only that the operation is performed on a limited region of the file. Argument *to* specifies the upper limit of the action range.

7.4.3.5 ***freadvector*** { vecspec <start next> }

Reads a vector from the file *infile* and assigns it to a programme's vector variable. *vecspec* is a specification of the variable to which the read vector is assigned. It must be given in a standard form consisting of the variable name and optionally the index table in square brackets. Reading starts at the current position in the file. A vector must reside at that position, otherwise the operation fails (blank characters are allowed between the current position and numbers of which the vector consists). The vector must be given as a sequence of numbers the first of which is vector's dimension and the others are its components. All the components must be given and must follow in a successive order. If the operation is successful, the current position in the file is set to the position of the first character after the read data. If it is partially successful, the current position is set to the position of the first character after the last read number.

An optional argument *start* specifies a name of the expression evaluator's variable to which the current position in the file before the operation is assigned, and an optional argument *next* specifies a name of the expression evaluator's variable to which the position of the first character after the read data is assigned. In the case that no data is read, the position before the operation is also assigned to the variable named *next*. These values are assigned to the expression evaluator's variables named *start* and *next* regardless of the success of the operation, therefore the success can not be examined through the values of these variables, but only through registered errors.

If the file is not defined, an error report is written to the standard output and to the programme's output file. If it is not connected to a physical file or if the vector can not be read, an appropriate error code is recorded. An error code is also recorded if only a part of the vector can be read.

7.4.3.6 ***freadvectorto*** { to vecspec <start next> }

Like **freadvector**, only that the operation is performed on a limited region of the file. Argument *to* specifies the upper limit of the action range.

7.4.3.7 ***freadmatrix*** { matspec <start next> }

Reads a matrix from the file *infile* and assigns it to a programme's matrix variable. *matspec* is a specification of the variable to which the read matrix is assigned. It must be given in a standard form consisting of the variable name and optionally the index table in square brackets. Reading starts at the current position in the file. A matrix must reside at that position, otherwise the operation fails (blank characters are allowed between the current position and numbers of which the matrix consists). The matrix must be given as a sequence of numbers the first two of which are matrix' dimensions and the others are its components. All the components must be given and must follow in a successive order (column indices changing quicker than row indices). If the operation is successful, the current position in the file is set to the position of the first character after the read data. If it is partially successful, the current position is set to the position of the first character after the last read number.

An optional argument *start* specifies a name of the expression evaluator's variable to which the current position in the file before the operation is assigned, and an optional argument *next* specifies a name of the expression evaluator's variable to which the position of the first character after the read data is assigned. In the case that no data is read, the position before the operation is also assigned to the variable named *next*. These values are assigned to the expression evaluator's variables named *start* and *next* regardless of the success of the operation, therefore the success can not be examined through the values of these variables, but only through registered errors.

If the file is not defined, an error report is written to the standard output and to the programme's output file. If it is not connected to a physical file or if the matrix can not be read, an appropriate error code is recorded. An error code is also recorded if only a part of the matrix can be read.

7.4.3.8 ***freadmatrixto*** { to matspec <start next> }

Like **freadmatrix**, only that the operation is performed on a limited region of the file. Argument *to* specifies the upper limit of the action range.

7.5 File Interpreter Functions of the Interface which Operate on Arbitrary Files

7.5.1 Controlling the State of the Files

7.5.1.1 ***fprintfilepos*** { filespec }

Prints a report about the current position in the file specified by *filespec* to the shell's output file.

7.5.1.2 ***printfilepos*** { filespec }

Prints a report about the current position in the file specified by *filespec* to the standard output.

7.5.1.3 ***dprintfilepos*** { filespec }

Prints a report about the current position in the file specified by *filespec* to the standard output and the programme's output file.

7.5.1.4 ***fprintfilepart*** { filespec pos1 <pos2> }

Prints a part of the file specified by *filespec* to the shell's output file. If *pos2* is specified, the part of the file from position *pos1* to (including) *pos2* is printed. If only *pos1* is specified, *pos1* bytes from the current position on (including the current position) of the file is printed. It prints the file part in the form of a report where it also prints which part of the file is printed.

If both *pos1* and *pos2* are specified, they can be zero, and *pos2* can be greater than the length of the file. In this case, if *pos1* is zero, it is changed to 1, if *pos2* is zero, it is changed to the length of the file, and if *pos2* is greater than the length of the file, it is also changed to the length of the file before the operation is performed. If only *pos1* is specified, it must be greater than zero. If in this case *pos1* bytes from the current position would exceed the length of the file, *pos1* is reduced before the operation so that the length of the file is matched.

An error is recorded if *pos1* or *pos2* are invalid. If *filespec* is not specified or the appropriate file variable is not defined, an error report is written to the standard output and to the programme's output file. If it is not connected to a physical file, an appropriate error code is recorded.

7.5.1.5 ***printfilepart*** { filespec pos1 <pos2> }

Does the same as **fprintfilepart**, except that it prints to standard output.

7.5.1.6 ***dprintfilepart*** { filespec pos1 <pos2> }

Does the same as **fprintfilepart**, except that it prints to the standard output and the programme's output file.

7.5.1.7 fwritefilepart { filespec pos1 <pos2> }

Prints a part of the file specified by *filespec* to the shell's output file. If *pos2* is specified, the part of the file from position *pos1* to (including) *pos2* is printed. If only *pos1* is specified, *pos1* bytes from the current position on (including the current position) of the file is printed. Only the file part is printed without any explanation or newlines, so this function is useful if the user wants to use the contents of a file as a part of the output which he generates on his own way.

If both *pos1* and *pos2* are specified, they can be zero, and *pos2* can be greater than the length of the file. In this case, if *pos1* is zero, it is changed to 1, if *pos2* is zero, it is changed to the length of the file, and if *pos2* is greater than the length of the file, it is also changed to the length of the file before the operation is performed. If only *pos1* is specified, it must be greater than zero. If in this case *pos1* bytes from the current position would exceed the length of the file, *pos1* is reduced before the operation so that the length of the file is matched.

An error is recorded if *pos1* or *pos2* are invalid. If *filespec* is not specified or the file variable is not defined, an error report is written to the standard output and to the programme's output file. If it is not connected to a physical file, an appropriate error code is recorded.

7.5.1.8 *writefilepart* { filespec pos1 <pos2> }

Does the same as **fwritefilepart**, except that it prints to standard output.

7.5.1.9 *dwritefilepart* { filespec pos1 <pos2> }

Does the same as **fwritefilepart**, except that it prints to the standard output and the programme's output file.

7.5.1.10 *filesetpos* { filespec pos }

Sets the current position in the file specified by *filespec* to the value of *pos*. *pos* should be a non-negative number not greater than the file size. If it is not an integer, it is rounded to the nearest integer. If it is zero, it is changed to the length of the file plus 1 before the operation is performed.

If it *pos* is greater than the length of the file, an error is recorded and the position is set to the end of the file (file length plus 1). If rounded *pos* is less than zero, an error is recorded and nothing else happens. If *fiespec* is not specified or the appropriate file variable is not defined, an error report is written to the standard output and to the programme's output file. If it is not connected to a physical file, an appropriate error code is recorded.

7.5.1.11 fileincreasepos { filespec inc }

Increases the current position in the file specified by *filespec* by *inc*. *inc* is rounded to the nearest integer. It can be negative (in this case the current position is decreased) or zero.

If the new position would be less than 1, an error code is recorded and the current position is set to 1. If the new position would be greater than the file length, an error code is recorded and the current position is set to the end of the file (the length of the file plus 1).

If *filespec* is not specified or the file variable is not defined, an error report is written to the standard output and to the programme's output file. If it is not connected to a physical file, an appropriate error code is recorded.

7.5.1.12 *filemarkpos* { filespec pos }

Marks the current position in the file specified by *filespec*. The position is assigned to the expression evaluator's variable named *pos*. If the position can not be determined, an integer value less than 1 is assigned to the variable.

If *filespec* is not specified or the appropriate file variable is not defined, an error report is written to the standard output and to the programme's output file. An error is not reported or recorded if the file is not connected to a physical file. In this case the marked position is less than one.

7.5.1.13 *fileflush* { filespec }

Empties the file buffer of the pre-defined file specified by *filespec* and writes the unsaved data on the disk.

This function is used only in special ocasions, for example when we debug the command file and would like to check the effect of file operations directly by editing a file. by executing the fflush command we make sure that all performed file operations take effect, since files are buffered and many operations only take immediate effect on the buffer, not on the file.

7.5.2 Searching for the Data

7.5.2.1 *filefindstring* { filespec string <pos after> }

Searches for the string *string* in the file specified by *filespec* from the current position on. If the string is found, it sets the current position of this file to the first byte of the found string. *pos* and *after* are optional arguments which specify the names of expression evaluator's variables into which additional information is stored. The function assigns the position of the string to the variable named *pos* and the position of the first byte after the string to the variable named *after*.

If *filespec* is not given or the appropriate file is not defined, an error report is written to the standard output and to the programme's output file. If it is not connected to a physical file or if the string is not found, an appropriate error code is recorded.

7.5.2.2 *filefindstringto* { filespec to string <pos after> }

Like **filefindstring**, only that the operation is performed on a limited region of the file. Argument *to* specifies the upper limit of the action range.

7.5.2.3 fileskipstring { filespec string <after pos> }

Searches for the string *string* in the file specified by *filespec* from the current position on. If the string is found, it sets the current position of this file to the first byte after the found string. Optional argument *after* specifies the name of the expression evaluator's variable to which the function assigns this position, and *pos* specifies the variable to which this function assigns the position of the found string (i.e. the first byte if this string if the file).

If *filespec* is not given or the appropriate file is not defined, an error report is written to the standard output and to the programme's output file. If it is not connected to a physical file or if the string is not found, an appropriate error code is recorded.

7.5.2.4 fileskipstringto { filespec to string <after pos> }

Like **fileskipstring**, only that the operation is performed on a limited region of the file. Argument *to* specifies the upper limit of the action range.

7.5.2.5 *filemultfindstring* { filespec {string1 string2 string3 string4 ...} <which pos after> }

Searches for the first occurrence of any of the strings *string1*, *string2*, *string3*, etc. in the file specified by *filespec* from the current position on. If it finds any of these strings, it sets the current position to the position of the found string. Strings must be in curly brackets.

If argument *which* is given, the function assigns the ordinary number of the found string to the expression evaluator variable named *which*. If argument *pos* is given, the function assign the position of the found string to the expression evaluator's variable named *pos*. If *after* is also given, the function assign the position of the first byte after the found string to the variable with such name.

If *filespec* is not given or the appropriate file is not defined, an error report is written to the standard output and to the programme's output file. If it is not connected to a physical file or if the string is not found, an appropriate error code is recorded.

7.5.2.6 *filemultfindstringto* { filespec to {string1 string2 string3 string4 ...} <which pos after> }

Like **filemultfindstring**, only that the operation is performed on a limited region of the file. Argument *to* specifies the upper limit of the action range.

7.5.2.7 *filemultskipstring* { filespec {string1 string2 string3 string4 ...} <which after pos> }

Searches for the first occurrence of any of the strings *string1*, *string2*, *string3*, etc. in the file specified by *filespec* from the current position on. If it finds any of these strings, it sets the current position to the position of the first byte after the found string. Strings must be in curly brackets.

If argument *which* is given, the function assigns the ordinary number of the found string to the expression evaluator variable named *which*. If argument *after* is given, the function assign the position of the first byte of the found string to the expression evaluator's variable named *after*, and if *pos* is also specified, the function assign the position of the found string to the variable with such name.

If *filespec* is not given or the appropriate file is not defined, an error report is written to the standard output and to the programme's output file. If it is not connected to a physical file or if the string is not found, an appropriate error code is recorded.

7.5.2.8 *filemultskipstringto* { filespec to {string1 string2 string3 string4 ...} <which after pos> }

Like **filemultskipstring**, only that the operation is performed on a limited region of the file. Argument *to* specifies the upper limit of the action range.

7.5.2.9 *filefindcharacter* {filespec charstring <pos> }

Searches in the file specified by *filespec* from the current position for the first occurrence of a character included in the string *charstring*. If the search is successful then it sets the current position in the file to the position of the found character.

Optional argument *pos* specifies a name of the expression evaluator's variable to which the position of the found character is assigned. If the search is not successful then a number less than 1 is assigned to that variable.

If *filespec* is not given or the appropriate file is not defined, an error report is written to the standard output and to the programme's output file. If it is not connected to a physical file or if the character is not found, an appropriate error code is recorded.

7.5.2.10 *filefindcharacterto* { filespec to charstring <pos> }

Like **filefindcharacter**, only that the operation is performed on a limited region of the file. Argument *to* specifies the upper limit of the action range.

7.5.2.11 *fileskipcharacters* { filespec charstring <pos> }

Searches in the file specified by *filespec* from the current position for the first occurrence of a character not included in the string *charstring*. If the search is successful then it sets the current position in the file to the position of the found character.

Optional argument *pos* specifies a name of the expression evaluator's variable to which the position of the found character is assigned. If the search is not successful then a number less than 1 is assigned to that variable.

If *filespec* is not given or the appropriate file is not defined, an error report is written to the standard output and to the programme's output file. If it is not connected to a physical file or if the character is not found, an appropriate error code is recorded.

7.5.2.12 *fileskipcharactersto* { filespec to charstring <pos> }

Like **fileskipcharacters**, only that the operation is performed on a limited region of the file. Argument *to* specifies the upper limit of the action range.

7.5.2.13 *filefindblank* { filespec <pos> }

Searches in the file specified by *filespec* from the current position for the first occurrence of a blank character, i.e. space, tab, newline, carriage return or null character. If the search is successful then it sets the current position in the file to the position of the found character.

Optional argument *pos* specifies a name of the expression evaluator's variable to which the position of the found character is assigned. If the search is not successful then a number less than 1 is assigned to that variable.

If *filespec* is not given or the appropriate file is not defined, an error report is written to the standard output and to the programme's output file. If it is not connected to a physical file or if the character is not found, an appropriate error code is recorded.

7.5.2.14 *filefindblankto* { filespec to <pos> }

Like **filefindblank**, only that the operation is performed on a limited region of the file. Argument *to* specifies the upper limit of the action range.

7.5.2.15 *fileskipblanks* { filespec file <pos> }

Searches in the file specified by *filespec* from the current position for the first occurrence of a non-blank character (blank characters are the space, tab, newline, carriage return and null character). If the search is successful then it sets the current position in the file to the position of the found character.

Optional argument *pos* specifies a name of the expression evaluator's variable to which the position of the found character is assigned. If the search is not successful then a number less than 1 is assigned to that variable.

If *filespec* is not given or the appropriate file is not defined, an error report is written to the standard output and to the programme's output file. If it is not connected to a physical file or if the character is not found, an appropriate error code is recorded.

7.5.2.16 *fileskipblanksto* { filespec to <pos> }

Like **fileskipblanks**, only that the operation is performed on a limited region of the file. Argument *to* specifies the upper limit of the action range.

7.5.2.17 *filenextline* { filespec <pos> }

Sets the current position in the file specified by *filespec* to the beginning of the next line from the current position in this file. If such position is found, it sets the current position in this file to it.

Optional argument *pos* specifies a name of the expression evaluator's variable to which the position of the found position is assigned. If the search is not successful then a number less than 1 is assigned to that variable.

If *filespec* is not given or the appropriate file is not defined, an error report is written to the standard output and to the programme's output file. If it is not connected to a physical file or if the next line can not be located (e.g. when the current position is in the last line of the file), an appropriate error code is recorded.

7.5.2.18 *filefindbrac* { filespec bracstr <pos1 pos2> }

Searches in the file specified by *filespec* from the current position for the first occurrence of a closed bracket specified by *bracstr*. The first character of *bracstr* must be the character used for the opening bracket and the second character of the string must be the character used for the closing bracket. These characters may not be the same. Sequences which represent special characters can be used in this string. If the search is successful then the function sets the current position in the file to the position of the first character inside the bracket if the bracket contain any characters, otherwise it sets the current position to the position of the first character after the bracket.

Optional arguments *pos1* and *pos2* specify names of the expression evaluator's variables to which the position of the found opening and closing bracket is assigned. If the search is not successful then a number less than 1 is assigned to those variables.

If *filespec* is not given or the appropriate file is not defined, an error report is written to the standard output and to the programme's output file. If it is not connected to a physical file or if a bracket is not found, an appropriate error code is recorded. An error code is also recorded if a bracket is found but does not contain any characters, i.e. the closing bracket immediately follow the opening bracket.

7.5.2.19 *filefindbracto* { filespec to bracstr <pos1 pos2> }

Like **filefindbrac**, only that the operation is performed on a limited region of the file. Argument *to* specifies the upper limit of the action range.

7.5.2.20 *fileskipbrac* { filespec bracstr <pos1 pos2> }

Searches in the file specified by *filespec* from the current position for the first occurrence of a closed bracket specified by *bracstr*. The first character of *bracstr* must be the character used for the opening bracket and the second character of the string must be the character used for the closing bracket. These characters may not be the same. Sequences which represent special characters can be used in this string. If the search is successful then the function sets the current position in the to the position of the first character after the bracket.

Optional arguments *pos1* and *pos2* specify names of the expression evaluator's variables to which the position of the found opening and closing bracket is assigned. If the search is not successful then a number less than 1 is assigned to those variables.

If *filespec* is not given or the appropriate file is not defined, an error report is written to the standard output and to the programme's output file. If it is not connected to a physical file or if a bracket is not found, an appropriate error code is recorded.

7.5.2.21 *fileskipbracto* { filespec to bracstr <pos1 pos2> }

Like **fileskipbrac**, only that the operation is performed on a limited region of the file. Argument *to* specifies the upper limit of the action range.

7.5.2.22 *filefindnumber* { filespec < start next val > }

Searches in the file specified by *filespec* from the current position for the first occurrence of a string which can represent a number. The number can be written in any standard format in which numbers are written in text files. If a number is found, the current position in the file is set to the position of the first character of the string which represent the number.

Optional arguments *start*, *next* and *val* specify names of the expression evaluator's variables to which the position of the found number, the position of the first character after the number, and the value of the number are assigned, respectively. If the search is not successful then integer numbers less than 1 are assigned to *start* and *next*.

If *filespec* is not given or the appropriate file is not defined, an error report is written to the standard output and to the programme's output file. If it is not connected to a physical file or if a number is not found, an appropriate error code is recorded.

7.5.2.23 *filefindnumberto* { filespec to < start next val > }

Like **filefindnumber**, only that the operation is performed on a limited region of the file. Argument *to* specifies the upper limit of the action range.

7.5.2.24 *fileskipnumber* { filespec < next start val > }

Searches in the file specified by *filespec* from the current position for the first occurrence of a string which can represent a number. The number can be written in any standard format in which numbers are written in text files. If a number is found, the current position in the file is set to the position of the first character after the string which represent the number.

Optional arguments *next*, *start* and *val* specify names of the expression evaluator's variables to which the position of the first character after the number, the position of the found number, and the value of the number are assigned, respectively. If the search is not successful then integer numbers less than 1 are assigned to *next* and *start*.

If *filespec* is not given or the appropriate file is not defined, an error report is written to the standard output and to the programme's output file. If it is not connected to a physical file or if a number is not found, an appropriate error code is recorded.

7.5.2.25 ***fileskipnumberto*** { filespec to < next start val > }

Like **fileskipnumber**, only that the operation is performed on a limited region of the file. Argument *to* specifies the upper limit of the action range.

7.5.3 Reading the Data

7.5.3.1 ***filereadnumber*** { filespec varname <start next> }

Reads a number from the file specified by *filespec* and assigns its value to the expression evaluator's variable named *varname*. Reading starts at the current position in the file. A number must reside at that position, otherwise the operation fails (blank characters are allowed between the current position and a number). If the operation is successful, the current position in the file is set to the position of the first character after the read data.

An optional argument *start* specifies a name of the expression evaluator's variable to which the current position in the file before the operation is assigned, and an optional argument *next* specifies a name of the expression evaluator's variable to which the position of the first character after the read data is assigned. In the case that no data is read, the position before the operation is also assigned to the variable named *next*. These values are assigned to the expression evaluator's variables named *start* and *next* regardless of the success of the operation, therefore the success can not be examined through the values of these variables, but only through registered errors.

If *filespec* is not given or the appropriate file is not defined, an error report is written to the standard output and to the programme's output file. If it is not connected to a physical file or if the number can not be read, an appropriate error code is recorded.

7.5.3.2 ***filereadnumberto*** { filespec to varname <start next> }

Like **filereadnumber**, only that the operation is performed on a limited region of the file. Argument *to* specifies the upper limit of the action range.

7.5.3.3 ***filereadscalar*** { filespec scalspec <start next> }

Reads a number from the file specified by *filespec* and assigns it to a programme's scalar variable. *scalspec* is a specification of the variable to which the number is assigned. It must be given in a standard form consisting of the variable name and optionally the index table in square brackets. Reading starts at the current position in the file. A number must reside at that position, otherwise the operation fails (blank characters are allowed between the current position and a number). If the operation is successful, the current position in the file is set to the position of the first character after the read data.

An optional argument *start* specifies a name of the expression evaluator's variable to which the current position in the file before the operation is assigned, and an optional

argument *next* specifies a name of the expression evaluator's variable to which the position of the first character after the read data is assigned. In the case that no data is read, the position before the operation is also assigned to the variable named *next*. These values are assigned to the expression evaluator's variables named *start* and *next* regardless of the success of the operation, therefore the success can not be examined through the values of these variables, but only through registered errors.

If *filespec* is not given or the appropriate file is not defined, an error report is written to the standard output and to the programme's output file. If it is not connected to a physical file or if the scalar can not be read, an appropriate error code is recorded.

7.5.3.4 *filereadscalarto* { filespec to scalspec <start next> }

Like **filereadscalar**, only that the operation is performed on a limited region of the file. Argument *to* specifies the upper limit of the action range.

7.5.3.5 *filereadvector* { filespec vecspec <start next> }

Reads a vector from the file specified by *filespec* and assigns it to a programme's vector variable. *vecspec* is a specification of the variable to which the read vector is assigned. It must be given in a standard form consisting of the variable name and optionally the index table in square brackets. Reading starts at the current position in the file. A vector must reside at that position, otherwise the operation fails (blank characters are allowed between the current position and numbers of which the vector consists). The vector must be given as a sequence of numbers the first of which is vector's dimension and the others are its components. All the components must be given and must follow in a successive order. If the operation is successful, the current position in the file is set to the position of the first character after the read data. If it is partially successful, the current position is set to the position of the first character after the last read number.

An optional argument *start* specifies a name of the expression evaluator's variable to which the current position in the file before the operation is assigned, and an optional argument *next* specifies a name of the expression evaluator's variable to which the position of the first character after the read data is assigned. In the case that no data is read, the position before the operation is also assigned to the variable named *next*. These values are assigned to the expression evaluator's variables named *start* and *next* regardless of the success of the operation, therefore the success can not be examined through the values of these variables, but only through registered errors.

If *filespec* is not given or the appropriate file is not defined, an error report is written to the standard output and to the programme's output file. If it is not connected to a physical file or if the vector can not be read, an appropriate error code is recorded. An error code is also recorded if only a part of the vector can be read.

7.5.3.6 *filereadvectorto* { filespec to vecspec <start next> }

Like **filereadvector**, only that the operation is performed on a limited region of the file. Argument *to* specifies the upper limit of the action range.

7.5.3.7 ***filereadmatrix*** { filespec matspec <start next> }

Reads a matrix from the file specified by *filespec* and assigns it to a programme's matrix variable. *matspec* is a specification of the variable to which the read matrix is assigned. It must be given in a standard form consisting of the variable name and optionally the index table in square brackets. Reading starts at the current position in the file. A matrix must reside at that position, otherwise the operation fails (blank characters are allowed between the current position and numbers of which the matrix consists). The matrix must be given as a sequence of numbers the first two of which are matrix' dimensions and the others are its components. All the components must be given and must follow in a successive order (column indexes changing quicker than row indexes). If the operation is successful, the current position in the file is set to the position of the first character after the read data. If it is partially successful, the current position is set to the position of the first character after the last read number.

An optional argument *start* specifies a name of the expression evaluator's variable to which the current position in the file before the operation is assigned, and an optional argument *next* specifies a name of the expression evaluator's variable to which the position of the first character after the read data is assigned. In the case that no data is read, the position before the operation is also assigned to the variable named *next*. These values are assigned to the expression evaluator's variables named *start* and *next* regardless of the success of the operation, therefore the success can not be examined through the values of these variables, but only through registered errors.

If *filespec* is not given or the appropriate file is not defined, an error report is written to the standard output and to the programme's output file. If it is not connected to a physical file or if the matrix can not be read, an appropriate error code is recorded. An error code is also recorded if only a part of the matrix can be read.

7.5.3.8 ***filereadmatrixto*** { filespec to matspec <start next> }

Like **filereadmatrix**, only that the operation is performed on a limited region of the file. Argument *to* specifies the upper limit of the action range.

7.5.4 Writing to Files and Copying File Parts

7.5.4.1 ***filewrite*** { filespec writespec }

Writes specified data in a specified format to the file specified by *filespec*. *filespec* is the specification of a file which consists of the file variable name and indexes of the file in the element table of this variable. *writespec* specifies what to write and in which form. The syntax of *writespec* is the same as at the functions *fwrite* and *write*. Writing begins at the current position of the file given by *filespec*. After the operation the current position is set to the first character after the written text.

An error report is written to the standard output and the programme's output file if *filespec* is not specified and if the specified file variable does not exist or if the corresponding file is not open.

7.5.4.2 ***copyfpart*** { targetspec pos1 <pos2> }

Copies a part of the file *infile* to the file specified by *targetspec*. *targetspec* is the specification of a file which consists of the file variable name and indexes of the file in the element table of this variable. If *pos2* is specified, the part of the file from position *pos1* to (including) *pos2* is copied. If only *pos1* is specified, *pos1* bytes from the current position on (including the current position) of the file is copied.

The current position of the file *infile* remains unchanged while the current position of the file specified by *targetspec* is set to the first byte after the last copied byte.

If both *pos1* and *pos2* are specified, they can be zero, and *pos2* can be greater than the length of the file. In this case, if *pos1* is zero, it is changed to 1, if *pos2* is zero, it is changed to the length of the file, and if *pos2* is greater than the length of the file, it is also changed to the length of the file before the operation is performed. If only *pos1* is specified, it must be greater than zero. If in this case *pos1* bytes from the current position would exceed the length of the file, *pos1* is reduced before the operation so that the length of the file is matched.

An error is recorded if *pos1* or *pos2* are invalid. If the file *infile* is not defined, an error report is written to the standard output and to the programme's output file. If it is not connected to a physical file, an appropriate error code is recorded.

7.5.4.3 ***copyfilepart*** { sourcespec targetspec pos1 <pos2> }

Copies a part of the file specified by *sourcespec* to the file specified by *targetspec*. *sourcespec* and *targetspec* are the specifications of files and consist of a file variable name and indexes of the file in the element table of the variable. If *pos2* is specified, the part of the file from position *pos1* to (including) *pos2* is copied. If only *pos1* is specified, *pos1* bytes from the current position on (including the current position) of the file is copied.

The current position of the file specified by *sourcespec* remains unchanged while the current position of the file specified by *targetspec* is set to the first byte after the last copied byte.

If both *pos1* and *pos2* are specified, they can be zero, and *pos2* can be greater than the length of the file. In this case, if *pos1* is zero, it is changed to 1, if *pos2* is zero, it is changed to the length of the file, and if *pos2* is greater than the length of the file, it is also changed to the length of the file before the operation is performed. If only *pos1* is specified, it must be greater than zero. If in this case *pos1* bytes from the current position would exceed the length of the file, *pos1* is reduced before the operation so that the length of the file is matched.

An error is recorded if *pos1* or *pos2* are invalid. If any of the files specified by *sourcespec* or *targerspec* is not defined, an error report is written to the standard output

and to the programme's output file. If any of these files is not connected to a physical file, an appropriate error code is recorded.

7.6 Expression Evaluator's Functions of the Interface

7.6.1.1 ***fileoperror*** [<which>]

If the function is called without arguments, it evaluates to the error status of the last file operation. Code 0 means that the last operation was performed successfully.

If the value of argument *which* is zero, the function evaluates to the number of recorded errors during file operations. Otherwise, the function evaluates to a specific error code where *which* is a successive number of an error on the stack of errors recorded during file operations. If the value of *which* is not integer, its value is rounded to the nearest integer. If it is negative, its absolute value specifies the successive number counted backwards (e.g. -1 refers to the last recorded error, -2 refers the pre-last recorded error, etc.).

7.6.1.2 ***getfpos*** [ ]

Evaluates to the current position in the file *infile*. If the current position is not defined (e.g. if the file is not existent) it evaluates to an integer number less than 1.

An error report is written to the standard output and the programme's output file if the file does not exist.

7.6.1.3 ***getfilepos*** [filename, < index1, index2, ... >]

Evaluates to the current position in the file specified by the name *filename* and indeces *index1*, *index2*, etc. If the current position is not defined (e.g. if the file is not existent) it evaluates to an integer number less than 1.

An error report is written to the standard output and the programme's output file if the file does not exist.

7.6.1.4 ***getfeof*** [ ]

Evaluates to a non-zero value if the current position in the file *infile* is at the end of the file, otherwise it evaluates to 0. It also evaluates to 0 if the current position can not be determined (e.g. if the file does not exist).

An error report is written to the standard output and the programme's output file if the file does not exist.

7.6.1.5 ***getfileeof*** [filename, < index1, index2, ... >]

Evaluates to a non-zero value if the current position in the file specified by the name *filename* and indexes *index1*, *index2*, etc., is at the end of the file, otherwise it evaluates to 0. It also evaluates to 0 if the current position can not be determined (e.g. if the file does not exist).

An error report is written to the standard output and the programme's output file if the file does not exist.

7.6.1.6 ***getflength*** [ ]

Evaluates to the length of the file *infile*. If the length can not be determined, it evaluates to an integer number less than 1 (e.g. if the file does not exist or is not connected to a physical file).

An error report is written to the standard output and the programme's output file if the file does not exist.

7.6.1.7 ***getfilelength*** [filename, < index1, index2, ... >]

Evaluates to the length of the file specified by the name *filename* and indexes *index1*, *index2*, etc. If the length can not be determined, it evaluates to an integer number less than 1 (e.g. if the file does not exist or is not connected to a physical file).

An error report is written to the standard output and the programme's output file if the file does not exist.

7.6.1.8 ***fcheckcharacter*** [charstr]

Evaluates to a non-zero number if the character on the current position in the file *infile* is contained in the string *charstr*, otherwise it evaluates to 0. Zero characters may not be contained in the *charstr*.

An error report is written to the standard output and the programme's output file if the file does not exist.

7.6.1.9 ***filecheckcharacter*** [filename, charstr, < index1, index2, ... >]

Evaluates to a non-zero number if the character on the current position in the file specified by the name *filename* and indexes *index1*, *index2*, etc., is contained in the string *charstr*, otherwise it evaluates to 0. Zero characters may not be contained in the *charstr*.

An error report is written to the standard output and the programme's output file if the file does not exist.

7.6.1.10 ***fcheckstring*** [str]

Evaluates to a non-zero number if the string *str* begins at the current position of the file *infile*, otherwise it evaluates to 0. It is not necessary that the string is followed by a blank character.

An error report is written to the standard output and the programme's output file if the file does not exist.

7.6.1.11*filecheckstring* [filename, str, < index1, index2, ... >]

Evaluates to a non-zero number if the string *str* begins at the current position of the file specified by the name *filename* and indexes *index1, index2*, etc., otherwise it evaluates to 0. It is not necessary that the string is followed by a blank character.

An error report is written to the standard output and the programme's output file if the file does not exist.

7.6.1.12*getfile* [varname spec <index1 index2 ...>]

Returns specific information about a file. *varname* is the name of the file variable and *index1, index2*, etc., are indexes which specify the file in the element table of the file variable in the case that the variable named *varname* contains more than one file. *spec* specifies which data should be returned. If the value of *spec* is zero, 0 is returned if the corresponding file is not open, otherwise a non-zero value is returned. If the value of *spec* is 1, 0 is returned if the file is not open for reading, otherwise a non-zero value is returned. If the value of *spec* is 2, 0 is returned if the file is not open for writing, otherwise a non-zero value is returned. If the value of *spec* is 3, the current position in the file is returned.

7.7 Interfacing the Analysis Input File

When the optimisation or inverse parameters are written explicitly somewhere in the input file for the simulation programme, we have a possibility of updating these parameters directly by a single command. First we must designate the numbers which represent the optimisation parameters in the direct analysis input file. The condition is that the analysis input file is a text file and that comment are allowed in the file, so that we can insert the mark that specify which numbers in the file represent the optimisation parameters.

The format of the mark is the following:

{ $$INV { num1 : param1 } { num2 : param2 } ... END }

num1, *num2*, etc. specify which numbers from the closing curly bracket on represent the specified parameters, and *param1*, *param2*, etc. specify which optimisation or inverse parameters correspond to these numbers. If, for example, *num1* is 3 and *param1* is 2, this means that the third number from the closed curly bracket on correspond to the second optimisation parameter. There can be unlimited marks in a single file and more than one number can correspond to the same parameter.

The parameter values are updated by the file interpreter functions **initinput** and **setparam**. The first function just finds the marks in the file and remembers which numbers in the file correspond to individual parameters. The second command updates the numbers in the file which correspond to parameters. These numbers are set to the

components of the pre-defined vector variable *parammom*. Both functions take no arguments and act on the pre-defined file *aninfile*.

Warnings:

The user must take care that the marks which specify the parameters in the analysis input file are inserted as comments, so that the meaning of the input file is not affected.

The user must leave enough space characters after the numbers that represent parameters because different numbers require different amount of characters for their representation in text format. The **setparam** function writes numbers directly without shifting the rest of the file in the case that the new value takes more or less space than the old one. The total amount of space available for the number representing a specific parameter must therefore exceed the maximum possible length of the numerical representation of a number.

Preserving additional space should not cause problems to the simulation programme when the input file is read. The functions which are used for reading the text input in most cases ignore spaces.

When the number that represents a specific parameter is marked, the user must take care at counting how many numbers are between the end of the mark and the target number. We must be aware that also parts of names that can be read as numbers must be taken into account. By the interface functions, all strings that can represent numbers are considered as numbers, even if they are only sub-strings of names, for example. The interface functions do not know the meaning of the input file contents. Therefore they consider as a number everything that loks like a number. For example, in the line

```
command1 3.45 12
```

the number 12 is considered to be the fourth number in the line. The first number is 1 that is a part of the string “command1”, the second number is 1 that is a part of the string “param1”, and the third number is 3.45.

Example:

Let’s say that we have the following lines somewhere in the analysis input file and the underlined numbers represent the first and the second parameter:

```
HARDENING  0  2  3
0.1 900
0.4 1050
0.7 1255
```

We can then mark the parameters in the following way:

```
* { $$INV {5:1} {7:2} END }
HARDENING  0  2  3
0.1 900
0.4 1050
0.7 1255
```

Here we supposed that the * sign means comment in the input file. The mark tells the interface functions that the fifth number after the mark represent the first optimisation parameter and the seventh number after the mark represents the second optimisation parameter. The mark could also be divided into two marks in the following way:

```
HARDENING  0  2  3
* { $$INV {2:1} END }
0.1 900
* { $$INV {2:2} END }
0.4 1050
0.7 1255
```

7.7.1 File Interpreter Functions for Interfacing the Analysis Input File

7.7.1.1 initinput *{ }*

The **initinput** function checks the file *aninfile* for marks which indicate the places where the optimisation parameters exist. It stores information about these places. This function must be executed before the **setparam** is called. The **setparam** function uses the information obtained by the **initinput** function to update the parameter values.

7.7.1.2 setparam *{ }*

The **setparam** function updates the values of optimisation parameters in the analysis input file *aninfile*. The values are taken from the vector variable *parammom*. The **initinput** function must be called before the **setparam** function to find the places in the file where the parameters reside. These places must be designated by special marks of the form

{ $$INV { num1 : param1 } { num2 : param2 } ... END }

The **initinput** function can be called only once for all **setparam** commands that follow.

7.7.1.3 setintfcdigits *{ numdigits }*

Sets the number of digits used while writing parameter values to the direct analysis input file *aninfile* by the function **setparam** to *numdigits*.

7.7.1.4 setintfccharacters *{ numcharacters }*

Sets the minimal length of output strings when writing parameter values to the direct analysis input file *aninfile* by the function **setparam** to *numcharacters*.

7.7.2 Expression Evaluator's Functions for Interfacing the Analysis Input File

7.7.2.1 getintfcdigits *[ ]*

Returns the number of digits used while writing parameter values to the direct analysis input file *aninfile* by the function **setparam**.

7.7.2.2 getintfccharacters *[ ]*

Returns the minimal length of output strings when writing parameter values to the direct analysis input file *aninfile* by the function **setparam**.

7.7.3 Setting Number of Digits at Number Output

Numbers are always output with certain precision. Number of significant digits that are output can be set by two functions, **setoutputdigits** and **setintfcdigits**. The first function affects general output functions like **write**, **fwrite**, **dwrite**, **printvector**, **fprintmatrix**, etc., while the second one affect interfacing functions like **setparam**.

7.8 Interaction with the File System

It is sometimes necessary that the shell can get information about the file system or interacts with it on some other way. In principle, this can always be done by using operating system commands running them by the **sustem** function. A set of functions described below enable this to be done in a more direct and system independent way. This includes functions for listing directories, changing the current directory and checking it, checking file properties, removing and deleting files, etc.

These functions must however be used with care. Not all of them are guaranteed to work on all operating systems. The most certainly they will work on Unix (including Linux) and MS Windows systems since they were developed for use on these systems in the first place. The second thing is that all differences between operating systems are not hidden by these functions. A typical example is using different characters for separating directories in long file names, e.g. '/' in Windows and '\' in Unix. Functions for interaction with the file system will generate file names compatible with the operating system conventions, while additional operations on these names (e.g. appending a file name to a directory name) can potentially be done using string manipulation functions. In

such cases the shell command file is not be transferable between different operating systems without slight changes.

7.8.1 File Interpreter Functions for Interaction with the File System

7.8.1.1 **checkfile** *{ filename varname1 < varname2 > }*

Checks status of the file named *filename* and assigns the result code to the expression evaluator variable named *varname1*. If the file is not readable, 0 is assigned, if the file is readable but not writable, 1 is assigned, and if the file is both readable and writable, 3 is assigned to the calculator variable named *varname1*. If the optional second variable name *varname2* is also specified, length of the file is assigned to the calculator variable named *var2* (respectively 0 if the file does not exist).

filename, *varname1* and *varname2* are string arguments.

7.8.1.2 **removefile** *{ filename }*

Removes the file named *filename*.

7.8.1.3 **renamefile** *{ filename1 filename2 }*

Renames the file names *filename1* to *filename2*.

7.8.1.4 ***appendtofile*** { filename writespec }

Writes specified data in a specified format to the end of the file on the disk named *filename*. This function avoids access to the file through a file variable and instead writes directly to the specified file. New content is appended to the old contents: if the file already exists, the old contents are not overwritten.

writespec specifies what to write and in which form. The syntax of *writespec* is the same as at the functions *filewrite*, *fwrite* and *write*.

7.8.1.5 **stringdirnames** *{ varname pathspec }*

Creates a list of files included in the path specification *pathspec* and stores it into a string variable named *varname*. Both *varname* and *pathspec* are string arguments.

File names are stored in a string variable of rank 1, whose dimension corresponds to the number of file names in the list. Previous contents are owerwritten if the variable has already existed. In either case the variable is created anew.

Remark:

This function is quite slow because it gets information through use of the system “list” or “dir” command.

7.8.1.6 stringdirdata *{ varname pathspec }*

Similar to **stringdirnames**, except that not only names, but also other information obtained by the appropriate system function is stored. A string variable of range two is therefore created to hold the data. Its first dimension equals the number of files in the list, and its second dimension equals the number of information pieces that are strored for each file.

Remark:

This function is quite slow because it gets information through use of the system "list" or "dir" command.

7.8.1.7 stringcurrentdir *{ elspec }*

Stores the name of the current directory to the string specified by *elspec*. The string is owerwritten it it has been initialised before function execution. If *elspec* contains any indices, the string element specified by *elspec* must exist. Otherwise, a string variable of rank zero is created and the name of the current directory assigned to its only element.

Remark:

This function is quite slow because it gets information through use of the system "list" or "dir" command.

7.8.1.8 changedir *{ pathspec }*

Changes the current directory to the directory specified by *pathspec*. *pathspec* is a string argument.

7.8.1.9 changedirwrite *{ arg1 arg2 … }*

Changes the current directory, where the directory name is specified by the combination of prints specified by *arg1 arg2*, etc. Arguments follow the same rules as arguments of **write**.

7.8.1.10 pwdinfo *{ }*

Prints information about the current directory (and host on some systems) to the standard output and programme output file. This function is usually used just to check if the current directory is the same as is expected.

7.8.1.11 stringtempfile *{ elspec }*

Generates a unique file name and assigns it to a string element specified by *elspec*. This string is provided by the computer's operating system and can be used as a name of a temporary file. The operating system should provide that no other file has such name and that the file with such name can be open for writing.