

File operations library

TXV 003 41.02
Second edition
march 2008
subject to alternations

Changes history

Date	Edition	Change description
January 2008	1	First edition
March 2008	2	DiskInfo function description added

CONTENT

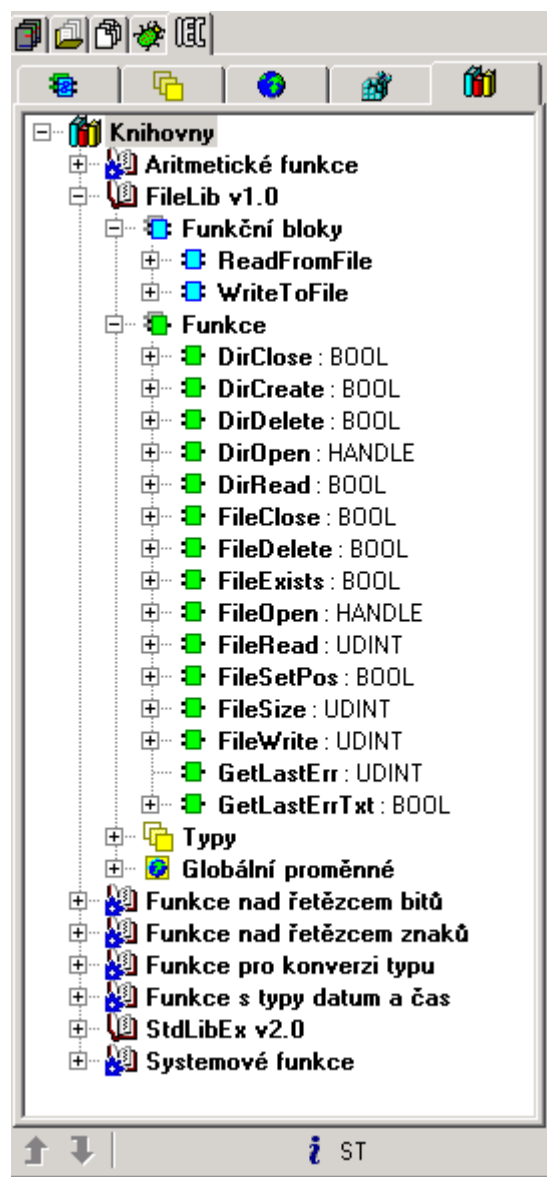
1 INTRODUCTION.....	5
2 DATA TYPES.....	8
2.1 HANDLE type.....	8
2.2 TfileInfo type.....	9
2.3 TF_MODE type.....	9
3 CONSTANTS.....	11
3.1 BEGIN_POS constant.....	11
3.2 END_POS constant.....	12
3.3 INVALID_HANDLE_VALUE constant.....	12
3.4 MAX_PATH constant.....	12
3.5 UNKNOWN_SIZE constant.....	13
4 FILE OPERATIONS FUNCTION	13
4.1 FileOpen function.....	14
4.2 FileRead function.....	16
4.3 FileWrite function.....	18
4.4 FileClose function.....	20
4.5 FileExists function.....	22
4.6 FileSize function.....	24
4.7 FileDelete function.....	26
4.8 FileSetPos function.....	28
4.9 DirOpen function.....	30
4.10 DirRead function.....	33
4.11 DirClose function.....	36
4.12 DirCreate function.....	38
4.13 DirDelete function.....	40
4.14 GetLastError function.....	42
4.15 GetLastErrorTxt function.....	43
4.16 DiskInfo function.....	44
5 FUNCTION BLOCK FOR FILE OPERATIONS.....	45
5.1 ReadFromFile function block.....	46
5.2 WriteToFile function block.....	50
5.3 The usage of ReadFromFile and WriteToFile function blocks.....	53

1 INTRODUCTION

The FileLib library contains a set of functions necessary for file operations. This library can be used for controlled systems equipped with file system.

The FileLib library contains basic (low-level) functions for file and directory operations, further, necessary data types and constants and last, function blocks for file reading and file entry. Function blocks use basic file functions and breakdown reading or file entry rather onto more cycles so, that the time consumption of these operations is optimized.

The following picture shows the FileLib library structure within the Mosaic environment.



If we want to use FileLib library functions in the PLC application program, we must firstly add this library to the project. The library is supplied as a part of Mosaic environment installation as from the version 2.6.0. The PLC central unit of the system must have the file operation support implemented.

Directories structure

The root directory for file operations within the PLC system is called ROOT. The system programmer can work only with such files and directories that are located in the ROOT directory. Other files and directories are not accessible from the PLC program. Therefore, the ROOT directory is the work directory for the PLC programmer.

File names

The file system supports file names in DOS 8.3 conversion. The file name consists of the file name itself (max. of 8 characters) and an appendix (max. of 3 characters). These two parts are separated by a dot. It is not possible to use punctuation characters, spaces and characters " * ", " ? " in file names. The national alphabet characters are not supported in file names. Capital and small letters are not distinguished. Substitutional characters (e. g. *.*)).

File path

The file path is a specification of a file location on the disc with regard to the ROOT directory. Therefore, the path contains names of directories where the file is saved. Same rules apply for names of directories within the path as for file name. Individual directory names within the path are divided by the slash sign „/“. The PLC file system supports absolute paths only. Relative paths nor even change of the work directory are not supported.

The maximum file name length inclusive of the path is limited by 65 characters (see the constant *MAX_PATH*).

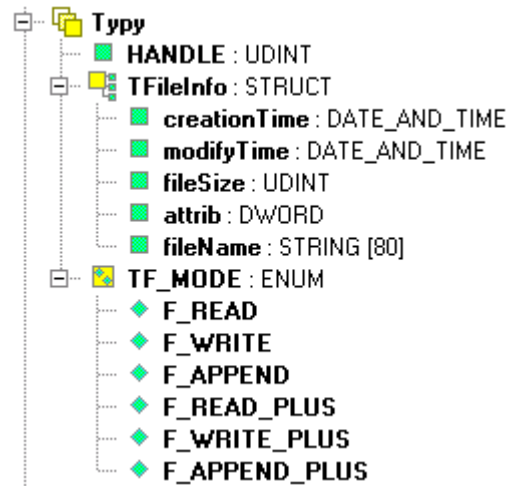
File visibility for the PLC web server

The PLC web server uses as a ROOT directory ROOT/WWW. If files should be accessible via the web interchange, they must be saved within this way (e. g. in the directory ROOT/WWW/DATA).

Details on supported file system type are shown in the description of relevant PLC central units.

2 DATA TYPES

In the FileLib library are following data types defined:



2.1 HANDLE type

HANDLE is a data type derived from the basic UDINT type and is used for a file identifier. Each open file has its identifier assigned that is used for file reading operations, file entry, file closing etc.

2.2 TFileInfo type

TFileInfo is a structure that is returned by the *DirOpen* and *DirRead* function and that contains file information.

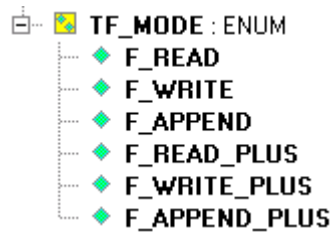


Meaning of individual items of the *TFileInfo* structure is as follows:

- *creationTime* date and time of file creation
- *modifyTime* date and time of the last file modification
- *fileSize* file size in bytes
- *attrib* file attributes
- *fileName* file name

2.3 *TF_MODE* type

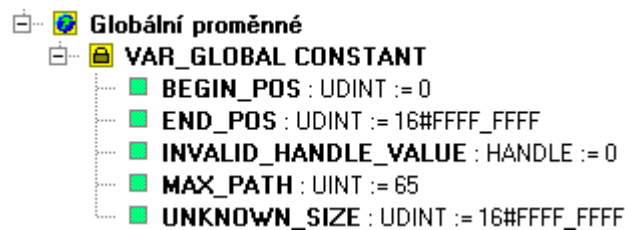
TF_MODE is an enumerational type that is used for constants defining the way of opening of the *FileOpen* function set.



Meaning of individual items of the enumeration, see function description *FileOpen*.

3 CONSTANTS

In the FileLib library the following constants are described:



3.1 *BEGIN_POS* constant

The *BEGIN_POS* constant is of an UDINT type and is used as a parameter of *FileSetPos* function in the case when we want to set reading or entry from the file beginning. The constant has a value of 0.

3.2 *END_POS* constant

The *END_POS* constant is of an UDINT type and is used as a parameter of the *FileSetPos* function in the case when we want to set reading or entry from the file end. The constant has a value of 16#FFFF_FFFF.

3.3 *INVALID_HANDLE_VALUE* constant

The *INVALID_HANDLE_VALUE* constant is of a HANDLE type and is used as a restoration value of *FileOpen* or rather *DirOpen* function in the case when the file or rather the directory failed to open. The constant has a value of 0 and means an invalid file identifier.

3.4 *MAX_PATH* constant

The *MAX_PATH* constant is of an UINT type and indicates the maximum possible length of file or directory name (path included). The constant has a value of 65.

3.5 *UNKNOWN_SIZE* constant

The *UNKNOWN_SIZE* constant is of an UDINT type and is used as a restoration value of *FileSize* function in the case when the file size could not be identified. The constant has a value of 16#FFFF_FFFF.

4 FILE OPERATIONS FUNCTION

The FileLib library contains following functions for file operations:

- *FileOpen* file opening
- *FileRead* file reading
- *FileWrite* file entry
- *FileClose* file closing
- *FileExists* file existence inquiry
- *FileSize* file size inquiry
- *FileDelete* file deleting
- *FileSetPos* file position setting

For directory operations the FileLib library contains following functions:

- *DirOpen* directory opening
- *DirRead* directory reading
- *DirClose* directory closing
- *DirCreate* directory creation
- *DirDelete* directory deleting

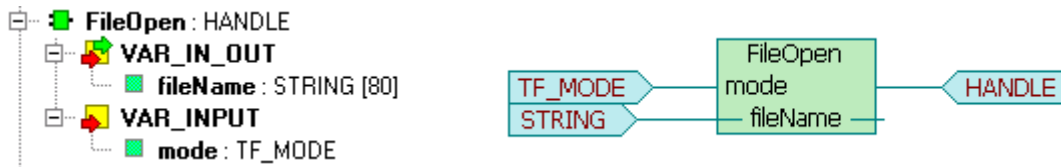
Other group of functions is functions enabling the specification of an error caused by and during file operations:

- *GetLastError* last error code inquiry
- *GetLastErrorTxt* error code transfer to the text report

And finally, function enabling disk information inquiry is available:

- *DiskInfo* disk size and disc free space inquiry

4.1 FileOpen function



FileOpen function opens the file specified by a variable *fileName*. The *mode* variable then states the required way of file access.

FileOpen function returns the identifier of the open file. If the file opening fails, this function restore the invalid identifier (INVALID_HANDLE_VALUE).

Variable description :

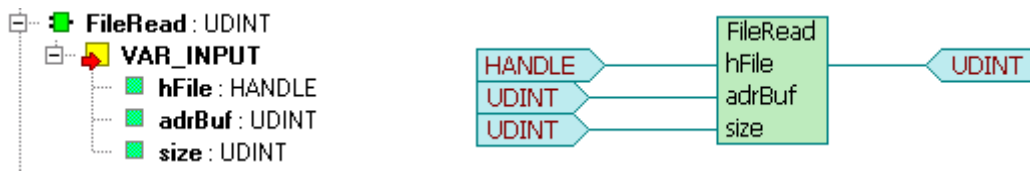
	Variable	Type	Signification
VAR_IN_OUT			
	<i>fileName</i>	STRING	File name including the path
VAR_INPUT			
	<i>mode</i>	TF_MODE	File access type
		F_READ	Open file for reading, the file must exist
		F_WRITE	Open file for entry data will be entered from the file beginning if the file does not exist, function establish a new file if the file exists, the file content is deleted
		F_APPEND	Open file for entry data will be entered to the file end if the file does not exist, function establish a new file if the file exists, data will be added at the file end
FileOpen			
	<i>Resto- ration value</i>	HANDLE	Open file identifier If the file opening fails, the invalid identifier is restored (INVALID_HANDLE_VALUE)

Example of the program with the FileOpen function call:

```
PROGRAM ExampleFileOpen
VAR
  path      : STRING := 'DATA/';
  name      : STRING := 'log.txt';
  hf        : HANDLE;           // file handle
END_VAR
VAR_TEMP
  result    : BOOL;
  fullFileName : STRING;
END_VAR

fullFileName := path + name;
hf := FileOpen(fileName := fullFileName, mode := F_READ);
IF hf <> INVALID_HANDLE_VALUE THEN
  result := FileClose( hFile := hf);
END_IF;
END_PROGRAM
```

4.2 FileRead function



The **FileRead** function load a certain number of characters and save them to the PLC memory. The file is specified by the *hFile* variable. The variable, where the loaded data will be saved to, is specified in *adrBuf*. And finally, the *size* variable states how many characters will be read from the file.

The **FileRead** function restores the number of really loaded characters.

Variable description :

	Variable	Type	Signification
VAR_INPUT			
	<i>hFile</i>	HANDLE	File identifier
	<i>adrBuf</i>	UDINT	Variable address where file character will be loaded to
	<i>size</i>	UDINT	Number of characters that should be loaded
FileRead			
	<i>Návratová hodnota</i>	UDINT	The number of really loaded characters

Example of the program with the FileRead function call :

```

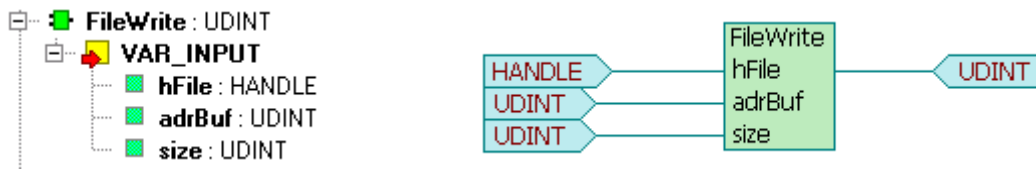
PROGRAM ExampleFileRead
VAR
    path      : STRING := 'DATA/';
    name      : STRING := 'log.txt';
    hf        : HANDLE;           // file handle
    buffer    : STRING[200];
    lenght    : UDINT;
END_VAR
VAR_TEMP
    result    : BOOL;
    fullFileName : STRING;
END_VAR

fullFileName := path + name;
hf := FileOpen(fileName := fullFileName, mode := F_READ);
IF hf <> INVALID_HANDLE_VALUE THEN
    lenght := FileRead( hFile := hf,
                      adrBuf := PTR_TO_UDINT( ADR(buffer)),
                      size := 100);
    result := FileClose( hFile := hf);
END_IF;

```

END_PROGRAM

4.3 FileWrite function



The **FileWrite** function copies the PLC variable content to the file. The file is specified by the *hFile* variable. The variable, where data will be copied from, is specified in *adrBuf*. And finally, the *size* variable states how many characters will be entered to the file.

The **FileWrite** function restores the number of really entered characters.

Variable description :

	Variable	Type	Signification
VAR_INPUT			
	<i>hFile</i>	HANDLE	File identifier
	<i>adrBuf</i>	UDINT	Variable address where content will be entered to the file
	<i>size</i>	UDINT	The number of characters entered
FileWrite			
	<i>Návratová hodnota</i>	UDINT	The number of really entered characters

Example of the program with the FileWrite function call :

```

PROGRAM ExampleFileWrite
VAR
  start   : BOOL;
  fname   : STRING := 'DATA/log.txt';
  hf      : HANDLE;           // file handle
  buffer  : STRING[200];
  lenght  : UDINT;
END_VAR
VAR_TEMP
  result  : BOOL;
END_VAR

IF start THEN
  start := FALSE;
  buffer := 'Hello world!';
  hf := FileOpen(fileName := fname, mode := F_WRITE);
  IF hf <> INVALID_HANDLE_VALUE THEN
    lenght := FileWrite( hFile  := hf,
                        adrBuf := PTR_TO_UDINT( ADR(buffer)),
                        size   := len( buffer));
    result := FileClose( hFile := hf);
  END_IF;

```

```
END_IF;  
END_PROGRAM
```

4.4 FileClose function



The **FileClose** function close the file specified by the *hFile* variable. All internal buffers for entry are prior to the file closing entered to the file. After the file closing, it is not possible to read from or entry to the file. The file identifier cease to be connected with the file.

The **FileClose** function restor the value TRUE, providing the file was closed successfully. Otherwise, it restores the value FALSE. The cause of a possible error can be found using function **GetLastError**.

Variable description :

	<i>Variable</i>	<i>Type</i>	<i>Signification</i>
VAR_INPUT			
	<i>hFile</i>	HANDLE	File identifier 
FileClose			
	<i>Návratová hodnota</i>	BOOL	TRUE if the file is  osed successfully, FALSE in other cases

Example of the program with the **FileClose** function call:

```

PROGRAM ExampleFileClose
VAR
  path          : STRING := 'DATA/';
  name          : STRING := 'log.txt';
  hf            : HANDLE;           // file handle
END_VAR
VAR_TEMP
  result        : BOOL;
  fullFileName : STRING;
END_VAR

fullFileName := path + name;
hf := FileOpen(fileName := fullFileName, mode := F_READ);
IF hf <> INVALID_HANDLE_VALUE THEN
  result := FileClose( hFile := hf);
END_IF;
END_PROGRAM

```

4.5 FileExists function



The **FileExists** function tests if the file specified by the *fileName* variable exists.

The **FileExists** function restore TRUE when the file exists. If the file search fails, this function restore FALSE.

Variable description :

	<i>Variable</i>	<i>Type</i>	<i>Signification</i>
VAR_IN_OUT			
	<i>fileName</i>	STRING	File name including the path
FileExists			
	<i>Návratová hodnota</i>	BOOL	TRUE if the file exists, FALSE in other cases

Example of the program with the FileExists function call :

```

PROGRAM ExampleFileExist
VAR
  fname   : STRING;
  del     : BOOL;
END_VAR

IF NOT del THEN
  fname := 'DATA/log.txt';
  IF FileExists( fileName := fname) THEN
    del := FileDelete( fileName := fname);
  END_IF;
END_IF;
END_PROGRAM

```

4.6 FileSize function



The **FileSize** function finds the file size specified by the variable *hFile*.

The **FileSize** function restores file size in bytes. If an error occurs during the file size evaluation, this function restores UNKNOWN_SIZE. The cause of a possible error can be found using function **GetLastError**.

Variable description :

	<i>Variable</i>	<i>Type</i>	<i>Signification</i>
VAR_INPUT			
	<i>hFile</i>	HANDLE	File identifier 
FileSize			
	<i>Restoration value</i>	UDINT	File size  UNKNOWN_SIZE, i.e. 16#FFFF_FFFF is restored when error occurs

Example of the program with the FileSize function call :

```

PROGRAM ExampleFileSize
VAR
  fname   : STRING;
  hf      : HANDLE;
  lenght  : UDINT;
  result  : BOOL;
END_VAR

fname := 'DATA/log.txt';
hf := FileOpen(fileName := fname, mode := F_READ);
IF hf <> INVALID_HANDLE_VALUE THEN
  lenght := FileSize(hFile := hf);
  result := FileClose( hFile := hf);
  result := result AND lenght <> UNKNOWN_SIZE;
END_IF;
END_PROGRAM

```

4.7 FileDelete function



The **FileDelete** function delete the file specified by the variable *fileName*.

This function can be used for directory deleting, too. To ensure succesful directory deleting, it is necessary to ensure that the directory is empty

The **FileDelete** function restores TRUE providing the file is deleted succesfully. Otherwise, function restores FALSE. The cause of a possible error can be found using function **GetLastError**.

Popis proměnných :

	Proměnná	Typ	Význam
VAR_IN_OUT			
	<i>fileName</i>	STRING	Jméno souboru včetně cesty
FileDelete			
	Návratová hodnota	BOOL	TRUE pokud je soubor smazán, FALSE v ostatních případech

Example of the program with the FileDelete function call :

```

PROGRAM ExampleFileDelete
VAR
  fname   : STRING;
  del     : BOOL;
END_VAR

IF NOT del THEN
  fname := 'DATA/log.txt';
  IF FileExists( fileName := fname) THEN
    del := FileDelete( fileName := fname);
  END_IF;
END_IF;
END_PROGRAM

```

4.8 FileSetPos function



The **FileSetPos** function enables to set the position for the file reading or rather for the file entry. The file is specified by the variable *hFile*. The required position in the file is stated by the variable *offset*.

The **FileSetPos** function restores TRUE providing that the required position was set successfully. Otherwise, it restores FALSE. The cause of a possible error can be found using function **GetLastError**.

Variable description :

Variable	Type	Signification
VAR_INPUT		
<i>hFile</i>	HANDLE	File identifier
<i>offset</i>	UDINT	Position in the file
		0 or BEGIN_POS File beginning
		0 > offset > END_POS Position set
		16#FFFF_FFFF or END_POS File end
FileSetPos		
<i>Restoration value</i>	BOOL	Restores TRUE if ; required position in the file is set Otherwise FALSE is restored

Example of the program with the FileSetPos function call:

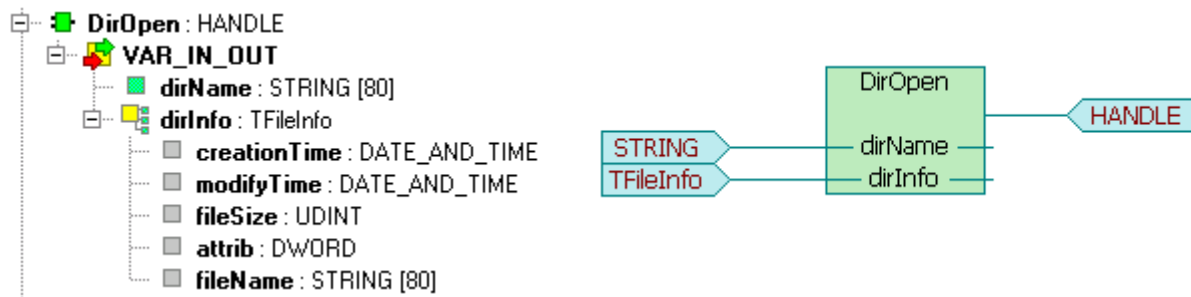
```

PROGRAM ExampleFileSetPos
VAR
  path      : STRING := 'DATA/';
  name      : STRING := 'log.txt';
  hf        : HANDLE;           // file handle
  buffer    : STRING[200];
  lenght    : UDINT;
END_VAR
VAR_TEMP
  result    : BOOL;
  fullFileName : STRING;
END_VAR

fullFileName := path + name;
hf := FileOpen(fileName := fullFileName, mode := F_READ);
IF hf <> INVALID_HANDLE_VALUE THEN
  IF FileSetPos( hFile := hf, offset := 10) THEN
  
```

```
    lenght := FileRead( hFile := hf,
                        adrBuf := PTR_TO_UDINT( ADR(buffer)),
                        size   := 100);
    result := FileClose( hFile := hf);
END_IF;
END_IF;
END_PROGRAM
```

4.9 DirOpen function



The **DirOpen** function opens the directory specified by the variable *dirName*. Then it finds information about the first file in the directory and enter these information in to the *dirInfo*. Information on other files in the directory can be found using function **DirRead**.

The **DirOpen** function restore the open directory identifier. If the directory opening fails, function restores an invalid identifier (**INVALID_HANDLE_VALUE**).

Variable description:

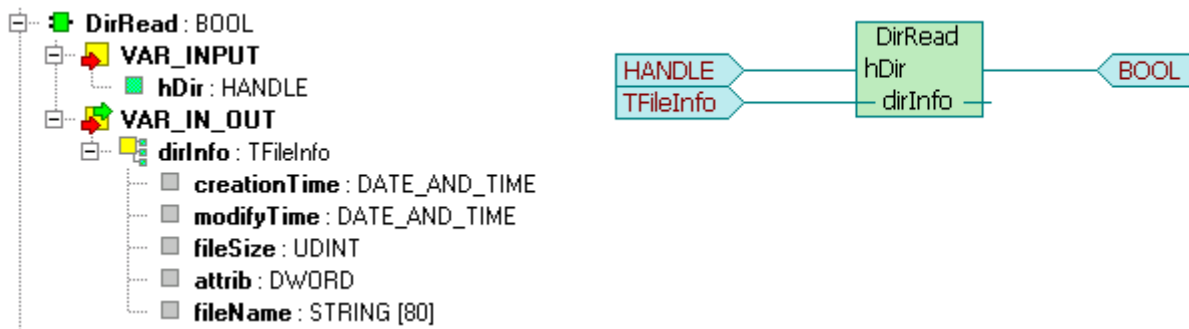
	Variable	Type	Signification
VAR_IN_OUT			
	<i>dirName</i>	STRING	Directory name including the path
	<i>dirInfo</i>	TFileInfo	Structure with information on the first file
	<i>.creationTime</i>	DT	Date and time of the file creation
	<i>.modifyTime</i>	DT	Date and time of the last file modification
	<i>.fileSize</i>	UDINT	File size
	<i>.attrib</i>	DWORD	File attributes
	<i>.fileName</i>	STRING	File name
DirOpen			
	<i>Restoration value</i>	HANDLE	Open directory identifier. If the opening fails, the invalid identifier is restored (INVALID_HANDLE_VALUE)

Example of the program with the DirOpen function call:

```
PROGRAM ExampleDirOpen
VAR
  dn      : STRING := 'DATA/';      // dir name
  hd      : HANDLE;                  // dir handle
  fi      : TFileInfo;
  result  : BOOL;
END_VAR

hd := DirOpen(dirName := dn, dirInfo := fi);
IF hd <> INVALID_HANDLE_VALUE THEN
  result := DirClose(hDir := hd);
END_IF;
END_PROGRAM
```

4.10 DirRead function



The **DirRead** function finds information about other file in the directory and enter this information in to the variable *dirInfo*. The directory is specified by a variable *dirHandle*. Before this function call, the directory must be open via the function *DirOpen*.

The **DirRead** function restores TRUE providing that the operation was successful. Otherwise, the FALSE value is restored.

Variable description :

Variable	Type	Signification
VAR_INPUT		
<i>dirHandle</i>	HANDLE	Directory identifier
VAR_IN_OUT		
<i>dirInfo</i>	TFileInfo	The structure with information on other file in the directory
<i>.creationTime</i>	DT	Date and time of file creation
<i>.modifyTime</i>	DT	Date and time of the last file modification
<i>.fileSize</i>	UDINT	File size
<i>.attrib</i>	DWORD	File attributes
<i>.fileName</i>	STRING	File name
DirRead		
<i>Restoration value</i>	BOOL	TRUE if operation was successful, FALSE in other cases

Example of the program with the DirRead function call:

```
TYPE
  T_FILE_INFO :
    STRUCT
      name : STRING[12];           // DOS name 8.3
      len  : UDINT;               // file length
    END_STRUCT;
END_TYPE

VAR_GLOBAL
  dirList : ARRAY[0..31] OF T_FILE_INFO;
END_VAR

PROGRAM ExampleDirRead
  VAR
    dn      : STRING := 'DATA/';  // dir name
    hd      : HANDLE;             // dir handle
    di      : TFileInfo;           // dir info
    count   : USINT;             // number of files
    done    : BOOL;
    err     : BOOL;
    open    : BOOL;
    result  : BOOL;
  END_VAR

  IF NOT done THEN
    IF NOT open THEN
      hd := DirOpen( dirName := dn, dirInfo := di);
      IF hd <> INVALID_HANDLE_VALUE THEN
        count := 1; open := TRUE;
        dirList[0].name := di.fileName;
        dirList[0].len  := di.fileSize;
      ELSE
        count := 0; err := TRUE; done := TRUE;
      END_IF;
    ELSE
      IF DirRead( hDir := hd, dirInfo := di) THEN
        dirList[count].name := di.fileName;
        dirList[count].len  := di.fileSize;
        count := count + 1;
      ELSE
        done := TRUE; err := FALSE; open := FALSE;
        result := DirClose( hDir := hd);
      END_IF;
    END_IF;
  END_IF;
END_PROGRAM
```

4.11 DirClose function



The **DirClose** function close the directory specified by the variable *hDir*. After the directory was closed, it is not possible to read from it using the function *ReadDir*. The directory identifier will cease to be connected to the file.

The **DirClose** function restores the value TRUE if the directory was closed successfully. Otherwise, it restores the value FALSE. The cause of a possible error can be found using the function **GetLastError**.

Variable description :

	<i>Variable</i>	<i>Type</i>	<i>Signification</i>
VAR_INPUT			
	<i>hFile</i>	HANDLE	File identifier 
DirClose			
	<i>Resto- ration value</i>	BOOL	TRUE if the direct  y was closed successfully, FALSE in other cases

Example of the program with the DirClose function call:

```

PROGRAM ExampleDirClose
VAR
  dn      : STRING := 'DATA/';      // dir name
  hd      : HANDLE;                  // dir handle
  fi      : TFileInfo;
  result  : BOOL;
END_VAR

hd := DirOpen(dirName := dn, dirInfo := fi);
IF hd <> INVALID_HANDLE_VALUE THEN
  result := DirClose(hDir := hd);
END_IF;
END_PROGRAM

```

4.12 DirCreate function



The **DirCreate** function creates the directory specified by the *dirName* variable.

The **DirCreate** function restores the value TRUE providing that the directory was created successfully. Otherwise, the value FALSE is restored. The cause of the possible error can be found using the function **GetLastError**.

Variable description :

Variable	Type	Signification
VAR_IN_OUT		
<i>dirName</i>	STRING	File name including the path
DirCreate		
Restoration value	BOOL	TRUE if the directory was created successfully, FALSE in other cases

Example of the program with the DirCreate function call:

```

PROGRAM ExampleDirCreate
VAR
    dn      : STRING := 'NEW/';      // dir name
    result  : BOOL;                  // dir exists
END_VAR

IF NOT FileExists( fileName := dn) THEN
    result := DirCreate( dirName := dn);
ELSE
    result := TRUE;
END_IF;
END_PROGRAM

```

4.13 DirDelete function



The **DirDelete** function deletes the directory specified by the variable *dirName*. To ensure a successful directory deleting, it is necessary that the directory is empty (it should not contain any files).

The **DirDelete** function restores TRUE if the directory is deleted successfully. Otherwise, FALSE is restored. The cause of the possible error can be found using the function **GetLastError**.

Variable description :

	Variable	Type	Signification
VAR_IN_OUT			
	<i>fileName</i>	STRING	Directory name including the path
DirDelete			
	<i>Restoration value</i>	BOOL	TRUE if the directory is deleted, FALSE in other cases

Example of the program with the DirDelete function call:

```

PROGRAM ExampleDirDelete
VAR
  dn      : STRING := 'NEW/';      // dir name
  fi      : TFileInfo;
  dh      : HANDLE;
  result  : BOOL;
END_VAR

IF FileExists( fileName := dn) THEN
  dh := DirOpen( dirName := dn, dirInfo := fi);
  IF dh = INVALID_HANDLE_VALUE THEN
    // directory is empty
    result := DirDelete( dirName := dn);
  ELSE
    // directory is not empty
    // it means we can't delete dir
    result := DirClose(hDir := dh);
  END_IF;
END_IF;
END_PROGRAM

```

4.14 GetLastErr function

 **GetLastErr** : UDINT



The **GetLastErr** function restores the code of the last error identified arisen during the file operation. This code can be used as a parameter of function **GetLastErrTxt** which then returns the text description of the error.

Variable description:

	<i>Variable</i>	<i>Type</i>	<i>Signification</i>
GetLastErr			
	<i>Restoration value</i>	UDINT	The code of the last  error within the file operation

Example of the program with the **GetLastErr** function call:

```

PROGRAM ExampleGetLastErr
VAR
  fname      : STRING;
  error      : UDINT;
  errTxt     : STRING;
  valid      : BOOL;
END_VAR

fname := 'DATA/log.txt';
IF NOT FileExists( fileName := fname) THEN
  error := GetLastErr();
  valid := GetLastErrTxt(errCode := error, errMessage := errTxt);
END_IF;
END_PROGRAM
  
```

4.15 GetLastErrTxt function



The **GetLastErrTxt** function enters the text description of an error in to the variable *errMsg*. The text description of an error corresponds to the error code that is specified by the variable *errCode*.

The **GetLastErrTxt** function restores TRUE if a text description of an error exists in comply with the set code. Otherwise, it restores FALSE.

Variable description :

Variable	Tye	Signification
VAR_INPUT		
<i>errCode</i>	UDINT	Error code
VAR_IN_OUT		
<i>errMsg</i>	STRING	Error text description
GetLastErrTxt		
<i>Restoration value</i>	BOOL	Restores TRUE if error description exists Otherwise, FALSE is restored

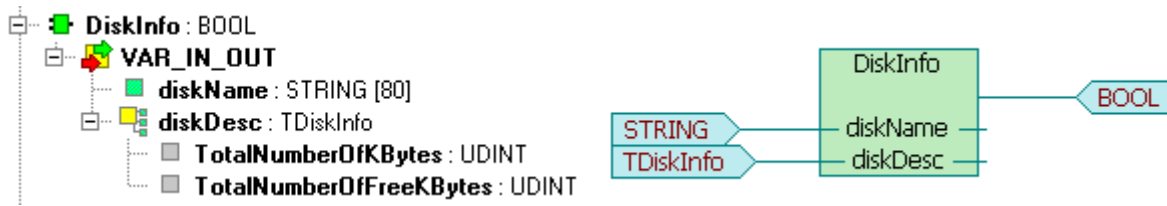
Example of the program with the GetLastErrTxt function call:

```

PROGRAM ExampleGetLastErrTxt
VAR
  fname    : STRING;
  error    : UDINT;
  errTxt   : STRING;
  valid    : BOOL;
END_VAR

fname := 'DATA/log.txt';
IF NOT FileExists( fileName := fname) THEN
  error := GetLastErr();
  valid := GetLastErrTxt(errCode := error, errMsg := errTxt);
END_IF;
END_PROGRAM
  
```

4.16 DiskInfo function



The **DiskInfo** function finds the overall disc size and free space on the disc. The disc name must be stated in the variable *diskName*. If this variable is empty, the **DiskInfo** function restores information about disc that is set as a default disc (e. g. it is a memory card within Foxtrot systems).

The **DiskInfo** function restores TRUE if information about required disc are found. Otherwise, it restores FALSE.

The **DiskInfo** function fill in items *TotalNumberOfKBytes* and *TotalNumberOfFreeKBytes* in the variable *diskDesc*. Both values are in KiloBytes.

Variable description :

Variable	Type	signification
VAR_IN_OUT		
<i>diskName</i>	STRING	Disc name
<i>diskDesc</i>	TDiskInfo	Disc information
. <i>TotalNumberOfKBytes</i>	UDINT	Total disc size in KB
. <i>TotalNumberOfFreeKBytes</i>	UDINT	Total free space on the disc in KB
DiskInfo		
<i>Restoration value</i>	BOOL	Restores TRUE if disc information are found. Otherwise, FALSE is restored

Example of the program with the DiskInfo function call:

```

PROGRAM ExampleDiskInfo
VAR
  disc    : STRING := '';      // default disk
  info    : TDiskInfo;        // info about disk
  result  : BOOL;
END_VAR

result := DiskInfo(diskName := disc, diskDesc := info);
END_PROGRAM

```

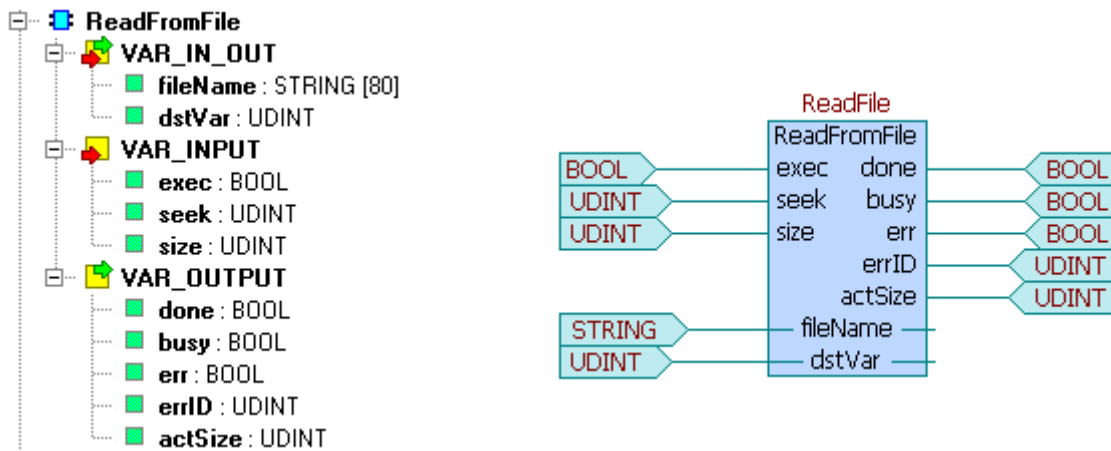
5 FUNCTION BLOCK FOR FILE OPERATIONS

The FileLib library contains following function blocks used for file operations:

- *ReadFromFile* file data reading
- *WriteToFile* file data entry

Function blocks use the basic file function and breakdown the reading or rather file entry in to more cycles so, that the time-consumption of these operations is optimized.

5.1 ReadFromFile function block



The **ReadFromFile** function block reads data from the file and saves them in to the variable in the PLC memory. The file name from which will be read is set by the variable *fileName*. The variable address, where data will be saved to, is specified by the variable *dstVar*. The reading from the file will be initiated onto the entering edge of the variable *exec*. The variable *seek* states the position from the beginning of the file from which data will be loaded. Data reading will be ceased when all required data are loaded which size is specified by the variable *size* or if the end of the file is reached during the reading.

The **ReadFromFile** function block sets TRUE into the variable *done* in the moment when the last data block is loaded from the file. During the data loading the variable *done* has a value FALSE and the variable *busy* has a value TRUE. The number of actually loaded bytes is stated in the variable *actSize*. If the reading was faultless, *err* variable has a value FALSE, in case of an error, it has a value TRUE and in the variable *errID* the error code is saved. That can be used as an entering variable of function *GetLastErrTxt* to obtain the text description of the error occurred.

Variable description :

Variable	Type	Signification
VAR_IN_OUT		
<i>fileName</i>	STRING	File name including the path
<i>dstVar</i>	UDINT	Variable address where read data from the file will be saved to
VAR_INPUT		
<i>exec</i>	BOOL	Control variable. Transition on entering edge (transition from the FALSE value to the TRUE value) initiate the file reading
<i>seek</i>	UDINT	Offset from the beginning of the file which the reading is initiated from
		0 From the file beginning
		$seek < 0$ From the set position

	<i>Variable</i>	<i>Type</i>	<i>Signification</i>
	<i>size</i>	UDINT	Required size of data read
VAR_OUTPUT			
	<i>done</i>	BOOL	Has a TRUE value at the moment of file reading ceasation Otherwise, returns FALSE
	<i>busy</i>	BOOL	Has a TRUE value during file data reading
	<i>err</i>	BOOL	Error flag within file reading If the operation was successful it has a FALSE value, otherwise, TRUE.
	<i>errID</i>	UDINT	Error code. If the operation was successful, errID = 0. In cas of an error, errID <> 0.
	<i>actSize</i>	UDINT	The number of actually loaded bytes

The following command shows the definition of the function block *ReadFromFile* in the ST language which is described here for the purpose of a detail understanding of this function. The usage of this block shows an example in the following chapter then.

```

FUNCTION_BLOCK ReadFromFile
(* Copy data from file to variable *)
VAR_IN_OUT
  fileName      : STRING;      // file name
  dstVar        : UDINT;       // destination variable
END_VAR
VAR_INPUT
  exec          : BOOL;        // request
  seek          : UDINT;       // data offset in file
  size          : UDINT;       // data length <= size of variable
END_VAR
VAR_OUTPUT
  done          : BOOL;        // action is done
  busy          : BOOL;        // action in progress
  err           : BOOL;        // error flag
  errID         : UDINT;       // error number
  actSize       : UDINT;       // really read
END_VAR
VAR
  execTrig      : R_TRIG;
  errTrig       : R_TRIG;
  hnd           : HANDLE;
  adrVar        : UDINT;
END_VAR
VAR_TEMP
  restSize      : UDINT;
  reqSize       : UDINT;
  read          : UDINT;
END_VAR

adrVar := PTR_TO_UDINT( ADR( dstVar));
execTrig( CLK := exec);

// open file and seek position

```

```

IF execTrig.Q THEN
  busy := TRUE; done := FALSE; err := FALSE; errID := 0; actSize := 0;
  errTrig( CLK := FALSE);
  hnd := FileOpen(fileName := fileName, mode := F_READ);
  IF hnd = INVALID_HANDLE_VALUE THEN
    err := TRUE; busy := FALSE;
  ELSE
    IF seek <> 0 THEN
      IF FileSetPos( hFile := hnd, offset := seek) = FALSE THEN
        err := TRUE; busy := FALSE;
      END_IF;
    END_IF;
  END_IF;
END_IF;

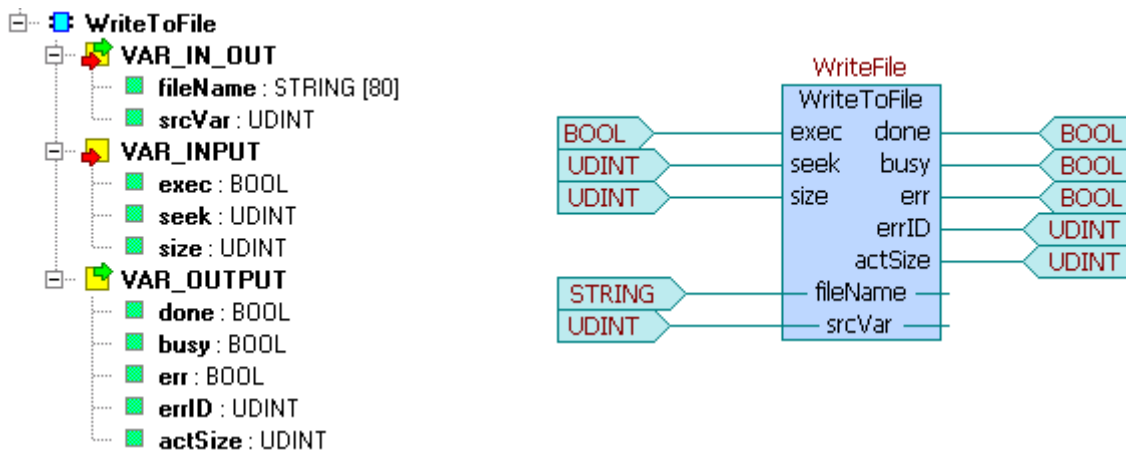
// read data from file to variable (one sector per one PLC cycle)
IF busy THEN
  IF size > actSize THEN
    restSize := size - actSize;
    IF restSize > 512 THEN reqSize := 512;           // more then one sector
    ELSE reqSize := restSize;           // last sector
  END_IF;
  read := FileRead( hFile := hnd, adrBuf := adrVar + actSize,
    size := reqSize);
  actSize := actSize + read;
  IF read <> reqSize THEN
    busy := FALSE;
    IF read > 0 THEN done := TRUE;           // end of file
    ELSE err := TRUE;           // propably any error
  END_IF;
  ELSE
    IF actSize = size THEN
      done := TRUE; busy := FALSE;
    END_IF;
  END_IF;
  ELSE
    done := TRUE;
  END_IF;
ELSE
  done := done AND exec;
  err := err AND exec;
END_IF;

// set error code, if any
errTrig( CLK := err);
IF errTrig.Q THEN errID := GetLastError(); END_IF;

// close file
IF errTrig.Q OR done THEN
  IF hnd <> INVALID_HANDLE_VALUE THEN
    IF FileClose( hFile := hnd) THEN
      hnd := INVALID_HANDLE_VALUE;
    END_IF;
  END_IF;
END_IF;
END_FUNCTION_BLOCK

```

5.2 WriteToFile function block



The **WriteToFile** function block enters the content of the PLC variable into the file. The file name where the entry will take place, states the variable *fileName*. If the file does not exist, it will be created. The address of the variable, the content of which will be entered to the file, is specified by the variable *srcVar*. The entry to the file will be initiated onto the entering edge of the variable *exec*. The *seek* variable states the position from the file beginning from where data will be entered. The data entry is ceased when all required data which size is stated by the variable *size* are entered..

The **WriteToFile** function block sets TRUE into the variable *done* at the moment when the last data block is entered to the file. During the data entry the variable *done* has a value FALSE and the variable *busy* has a value TRUE. The number of actually entered bytes is stated by the variable *actSize*. If the file entry was faultless, the *err* variable has a value FALSE. In case of an error occurrence it has a value TRUE and in the variable *errID* is saved the error code. This code can be used as an entering variable function *GetLastErrorTxt* to get the text description of the error arisen.

Variable description :

	Variable	Type	Signification
VAR_IN_OUT			
	<i>fileName</i>	STRING	File name including the path
	<i>srcVar</i>	UDINT	Address of the variable which content will be entered to the file
VAR_INPUT			
	<i>exec</i>	BOOL	Control variable. Entering edge (transition from FALSE value to TRUE value) initiates the file entry
	<i>seek</i>	UDINT	Offset in the file from which the entry is initiated
			0 From the file beginning
			0 < seek < 16#FFFF_FFFF From the set position
			16#FFFF_FFFF Add data to the file end

	<i>Variable</i>	<i>Type</i>	<i>Signification</i>
	<i>size</i>	UDINT	Required size of data entered
VAR_OUTPUT			
	<i>done</i>	BOOL	Has a TRUE value at the moment of the file entry ceasation Otherwise, it returns FALSE
	<i>busy</i>	BOOL	Has a TRUE value during the entry to the file
	<i>err</i>	BOOL	Error flag during file entry If the operation was successful, it has a FALSE value, otherwise, TRUE.
	<i>errID</i>	UDINT	Error code. If the creation was successful, errID = 0. In case of an error, errID < 0.
	<i>actSize</i>	UDINT	Number of actually entered bytes

The following command shows the definition of the function block *WriteToFile* in the ST language which is described here for the purpose of a detail understanding of this function. The usage of this block shows an example in the following chapter then.

```

FUNCTION_BLOCK WriteToFile
(*
  Copy data from variable to file.
  If file does not exist new file is created.
  If file exists file content is overwritten.
*)
VAR_IN_OUT
  fileName      : STRING;           // file name
  srcVar        : UDINT;           // variable address
END_VAR
VAR_INPUT
  exec          : BOOL;            // request
  seek          : UDINT;           // data offset in file
  size          : UDINT;           // data length (size of variable)
END_VAR
VAR_OUTPUT
  done          : BOOL;            // action is done
  busy          : BOOL;            // action in progress
  err           : BOOL;            // error flag
  errID         : UDINT;           // error number
  actSize       : UDINT;           // really written
END_VAR
VAR
  eTrig         : R_TRIG;
  errTrig       : R_TRIG;
  hnd           : HANDLE;
  adrVar        : UDINT;
END_VAR
VAR_TEMP
  restSize      : UDINT;
  written       : UDINT;
  reqSize       : UDINT;
  mode          : TF_MODE;
END_VAR

```

```

adrVar := PTR_TO_UDINT( ADR( srcVar));
eTrig(CLK := exec);

// open file and seek position
IF eTrig.Q THEN
  busy := TRUE; done := FALSE; err := FALSE; errID := 0; actSize := 0;
  errTrig( CLK := FALSE);
  IF seek = 0 THEN mode := F_WRITE; ELSE mode := F_APPEND; END_IF;
  hnd := FileOpen( fileName := fileName, mode := mode);
  IF hnd = INVALID_HANDLE_VALUE THEN
    err := TRUE; busy := FALSE;
  ELSE
    IF seek <> 0 AND seek <> 16#FFFF_FFFF THEN
      IF FileSetPos( hFile := hnd, offset := seek) = FALSE THEN
        err := TRUE; busy := FALSE;
      END_IF;
    END_IF;
  END_IF;
END_IF;

// write data to file to variable (one sector per one PLC cycle)
IF busy THEN
  IF size > actSize THEN
    restSize := size - actSize;
    IF restSize > 512 THEN reqSize := 512;           // more then one sector
    ELSE reqSize := restSize;           // last sector

    END_IF;
    written := FileWrite(hFile := hnd, adrBuf := adrVar + actSize,
                        size := reqSize);

    actSize := actSize + written;
    IF written <> reqSize THEN
      busy := FALSE; err := TRUE;
    ELSE
      IF actSize = size THEN
        done := TRUE; busy := FALSE;
      END_IF;
    END_IF;
  ELSE
    done := TRUE;
  END_IF;
ELSE
  done := done AND exec;
  err := err AND exec;
END_IF;

// set error code, if any
errTrig( CLK := err);
IF errTrig.Q THEN errID := GetLastError(); END_IF;

// close file
IF errTrig.Q OR done THEN
  IF hnd <> INVALID_HANDLE_VALUE THEN
    IF FileClose( hFile := hnd) THEN
      hnd := INVALID_HANDLE_VALUE;
    END_IF;
  END_IF;
END_IF;
END_FUNCTION_BLOCK

```

5.3 The usage of ReadFromFile and WriteToFile function blocks

The following example shows the usage of function blocks ReadFromFile and WriteToFile. The program, in the example, will test after the system restart whether the *DATA* directory exist. If it does not, it will create it. Afterwards, it creates the file *test.txt* in this directory and enters data saved in the variable *record* in to this file. Then, the content of the *test.txt* file is loaded into the variable *copy* that will compare it with the variable *record*. The content of both variables must be identical. So it is a simple test of data entry to the file and their loading for the purpose of the control. The test result is stated by variables *test_OK* and *test_ERR*. The reason of possible difficulties can be then found in the variable *problem*.

```
TYPE
  TRecord : STRUCT
    text  : STRING;
    data  : ARRAY[0..999] of USINT;
  END_STRUCT;
END_TYPE

VAR_GLOBAL
  test_OK   : bool;           // result
  test_ERR  : bool;           // error flag
  record    : TRecord;        // written data
  copy      : TRecord;        // read data
END_VAR

FUNCTION InitRecord : BOOL
  VAR i : UINT; END_VAR

  record.text := 'Begin of file DATA/test.txt ...';
  FOR i := 0 TO 999 DO record.data[i] := UINT_TO_USINT(i); END_FOR;
  InitRecord := TRUE;
END_FUNCTION

PROGRAM Test_MMC_card
  VAR
    dir,
    read, init          : BOOL;
    write               : BOOL;
    fn                  : STRING := 'DATA/test.txt';
    dn                  : STRING := 'DATA/';
    WriteFile           : WriteToFile;
    ReadFile            : ReadFromFile;
    count               : UINT;
    problem              : STRING := 'No problem';
  END_VAR

  IF NOT init THEN
    init := InitRecord();
    dir := FileExists( fileName := dn);
    IF NOT dir THEN
```

```
    dir := DirCreate(dirName := dn);
    IF NOT dir THEN
        test_ERR := GetLastErrTxt( errCode := GetLastErr(),
                                   errMessage := problem);
    END_IF;
END_IF;
END_IF;

IF dir THEN
    IF NOT WriteFile.err AND NOT ReadFile.err THEN
        IF NOT write THEN
            count := count + 1;
            WriteFile( fileName := fn, srcVar := void( record), exec := true,
                      seek := 0, size := sizeof( record), done => write);
            IF WriteFile.err THEN
                test_ERR := GetLastErrTxt( errCode := WriteFile.errID,
                                           errMessage := problem);
            END_IF;
        END_IF;
        IF write AND NOT read THEN
            count := count + 1;
            ReadFile( fileName := fn, dstVar := void( copy),
                    exec := true, seek := 0,
                    size := sizeof( copy), done => read);
            IF ReadFile.err THEN
                test_ERR := GetLastErrTxt( errCode := ReadFile.errID,
                                           errMessage := problem);
            END_IF;
        END_IF;
    END_IF;

    IF write AND read THEN
        IF record = copy THEN test_OK := true; END_IF;
    END_IF;
END_IF;
END_PROGRAM
```