Memory Cards – Tiger Basic API

V 1.3

Introduction

This document describes the Tiger Basic API (application programming interface) for SD/MMC and SmartMedia card.

The chapter “Configure Interface to Memory Cards Hardware” explains how to set up pins and addresses that are relevant for communication with attached SD/MMC card hardware and how to set the parameters of the SmartMedia device driver. To obtain the information about what pins and addresses are to use, see the corresponding datasheets.

The chapter “Direct Access to Memory Cards” describes the subroutines of the API that implement direct access to the card memory without involving any file/directory constructs.

The chapter “File System (FAT) Support for Memory Cards” concentrates on the work with the card using the FAT file system.

The access to SD/MMC cards is based on SPI and implemented by using of shift_in, shift_out instructions of the Tiger Basic programming language, without any special device driver.

The access to SmartMedia card is based on using of a dedicated device driver SMEDIA_128MB.

Change History of this Document

V1.2 to V1.3:

1. New chapter “Deactivate FAT subroutines” for Direct Access is added.
2. This chapter “Change History of this Document” is added

Configure interface to Memory Card hardware

Interface to SD/MMC hardware

The file sd_card_pins_d.inc contains the pin definitions for the different configurations.

The configurations define how the SD/MMC card is connected to the Tiger Basic.

The pin definition block is normally surrounded by "ifdef … endif" pair like:

```
' sd_card_pins_d.inc

'*************************************************************************
'                 Pin Description for My Own Configuration
'*************************************************************************

#ifdef MY_CONFIGURATION
. . . . . .
#endif  '' MY_CONFIGURATION
```

Such structure of the pin definition block enables very simple selecting of the active hardware configuration. The corresponding "define" instruction must merely appear in the TIG file before any "include" instruction.

```
' my_application.tig

' Activate My Own Configuration
#define MY_CONFIGURATION
. . . . . .
#include define_a.inc
#include fs_coinc.inc
. . . . . .
```

The following table shows what constants must be defined to execute the SD card routines correctly and what meaning the constants have.

Some service lines of the SD card hardware can be connected by using of either tiger module pins directly, or X-Port (see e.g. SD_XP_POWER and SD_I_POWER). Two constants are reserved for these lines, but only one of these constants must be active in the particular configuration.

Configuration Constant	Description
SD_SPI_PORT	Port for SPI lines (all three on one Port!)
SD_SPI_DATA_IN_PIN	Pin for SPI MiSo (Tiger-In, Card-Out)
SD_SPI_DATA_OUT_PIN	Pin for SPI MoSi (Tiger-Out, Card-In)
SD_SPI_CLOCK_PIN	Pin for SPI Clock
SD_XP_ADDRESS	X-Port Address for Control Lines (if it is not defined, all SD_XP_x settings are disabled)
SD_XP_POWER_OFFSET	Offset to X-Port Address for Power Line (if not necessary, must be set to zero)
SD_XP_SDCARD_DETECT	Bit of X-Port for Card Detect Line
SD_XP_WRITE_PROTECT	Bit of X-Port for Write Protect Line
SD_XP_POWER	Bit of X-Port for Power Line
SD_XP_ERROR_INDICATOR	Bit of X-Port for Error Indication Line
SD_XP_CHIP_SELECT	Bit of X-Port for Chip Select Line
SD_I_PORT	Port for Control Lines (if it is not defined, all SD_I_x settings are disabled)
SD_I_SDCARD_DETECT	Pin for Card Detect Line
SD_I_WRITE_PROTECT	Pin for Write Protect Line
SD_I_POWER	Pin for Power Line
SD_I_ERROR_INDICATOR	Pin for Error Indication Line
SD_I_CHIP_SELECT	Pin for Chip Select Line

The simplest way to create own pin definition block is to copy an existing block and modify it. For pre-defined pin definition blocks see the sd_card_pins_d.inc file.

Example of a pin definition block from the sd_card_pins_d.inc file.

```
'*************************************************************************'
'                        Pin Description for SD Card Adapter 1
'*************************************************************************'
#ifdef SD_ADAPTER1
'' SPI configuration
#define SD_SPI_PORT                             8
'' MiSo (Tiger - In, Card - Out)
#define SD_SPI_DATA_IN_PIN                      2
'' MoSi (Tiger - Out, Card - In)
#define SD_SPI_DATA_OUT_PIN                     1
'' Clock
#define SD_SPI_CLOCK_PIN                        0

'' Features controlled over X-Port
'' X-Port Address for Control Lines
#define SD_XP_ADDRESS                           00F8h
'' Offset to X-Port Address for Power Line (if not necessary, must be set to
zero)
#define SD_XP_POWER_OFFSET                      1

#ifdef SD_XP_ADDRESS
'' Bit of X-Port for Card Detect Line
#define SD_XP_SDCARD_DETECT                     2
'' Bit of X-Port for Write Protect Line
#define SD_XP_WRITE_PROTECT                     4
'' Bit of X-Port for Power Line
#define SD_XP_POWER                             6
'' Bit of X-Port for Error Indication Line (unused in this configuration)
'' #define SD_XP_ERROR_INDICATOR                1
'' Bit of X-Port for Chip Select Line (unused in this configuration)
'' #define SD_XP_CHIP_SELECT                    0
#endif  '' SD_XP_ADDRESS

'' Features controlled over Tiger-Pins
'' Port Address for Control Lines
#define SD_I_PORT                               8
#ifdef SD_I_PORT
'' Pin for Card Detect Line (unused in this configuration)
'' #define SD_I_SDCARD_DETECT                   2
'' Pin for Write Protect Line (unused in this configuration)
'' #define SD_I_WRITE_PROTECT                   4
'' Pin for Power Line (unused in this configuration)
'' #define SD_I_POWER                           7
'' Pin for Error Indication Line
#define SD_I_ERROR_INDICATOR                    3
'' Pin for Chip Select Line
#define SD_I_CHIP_SELECT                        6
#endif  '' SD_I_PORT
#endif  '' SD_ADAPTER1
```

Interface to SmartMedia hardware

The device driver SMEDIA_128MB implements the interface to SmartMedia hardware. This driver has to be installed in the Main task of a Tiger Basic program.

The following parameters of the SMEDIA_128MB driver can be set to configure the interface to SmartMedia hardware.

install_device #SMCARD, "SMEDIA_128MB.TDD", P1, P2, P3, P4, P5, P6, P7

P1	data bus port,
P2	control port,
P3	read enable pin of control port,
P4	write enable pin of control port,
P5	chip enable pin of control port,
P6	command latch enable pin of control port,
P7	ADR latch enable pin of control port.

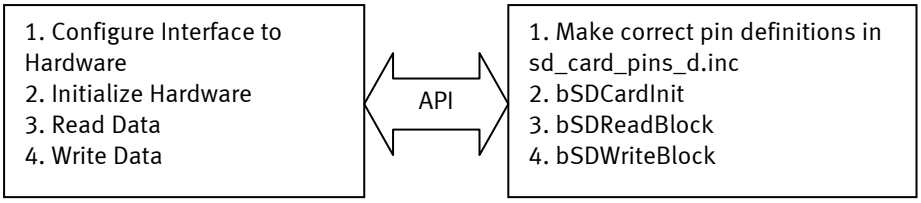
```
' install SmartMedia with expl. settings
'                                          Ctrl
'                                        Bus  RE WE CE CLE ALE, speed
  install_device #SMCARD, "SMEDIA_128MB.TDD",6,8,0, 1, 6,  4,  2
```

Direct Access to Memory Cards

Direct Access to SD/MMC Card

Working with SD/MMC Card using Direct Access

The following simple steps are necessary to read from or write to SD/MMC card using direct access methods.



Deactivate FAT subroutines

To reduce FLASH and RAM consumption, one can deactivate FAT subroutines that are normally not applied for direct access to the card. Commenting out the definition FAT_ACTIVE in the file "fs_conf.inc" excludes the FAT subroutines and global variables from compilation process.

Initialize hardware

sub bSDCardInit(var byte bpvRet)

bpvRet return value that is set to TRUE on success and to FALSE on error.

The subroutine bSDCardInit completes the following tasks:

- Initialization of the SPI lines and additional control lines connected to SD card interface,
- Checking the SD card detection line,
- Resetting the SD card,
- Initialization of the API global variables.

The subroutine bSDCardInit must be called before any other subroutine of this SD card API is called.

Example for the subroutine bSDCardInit:

```
byte blStatus
. . . . . .
call bSDCardInit( blStatus )
```

Read Data

sub bSDReadBlock(long lpAddress; long lpSize; var string spvBuffer$; var byte bpvResult)

lpAddress	the address on the card to read data from. The first address on the card is 0 (zero).
lpSize	the size of the data to read.
spvReadData$	the buffer to store the read data. The buffer must be large enough to hold all the read bytes.
bpvResult	return value that is set to TRUE on successful reading and to FALSE on error.

Example for the subroutine bSDReadBlock:

```
#define READ_DATA_SIZE  512
. . . . . .
byte blStatus
long llAddress, llSize
string slReadData$( READ_DATA_SIZE )
. . . . . .
llAddress = 0c600h
llSize = READ_DATA_SIZE
slReadData$ = ""
call bSDReadBlock (llAddress, llSize, slReadData$, blStatus)
```

Write Data

sub bSDWriteBlock(long lpAddress; long lpSize; var string spvDataToWrite$; var byte bpvResult)

lpAddress	the address on the card to write data to. The first address on the card is 0 (zero).
lpSize	the size of the data to write.
spvDataToWrite$	the buffer containing the data to write to the card.
bpvResult	return value that is set to TRUE on successful reading and to FALSE on error.

Example for the subroutine bSDWriteBlock:

```
#define DATA_TO_WRITE_SIZE      512
. . . . . .
byte blStatus
long llAddress, llSize
string slDataToWrite$( DATA_TO_WRITE_SIZE )
. . . . . .
llAddress = 0c600h
llSize = DATA_TO_WRITE_SIZE
slDataToWrite$ = “The quick brown fox jumps over the lazy dog”
call bSDWriteBlock (llAddress, llSize, slDataToWrite$, blStatus)
```

Read Card Size

sub lSDCardGetSize(var long lpvSDCardSize)

lpvSDCardSize the real size of the card.

The real size of the card can slightly differ from the size on the card label. To obtain the rounded size of the card, call the subroutine lSDCardGetRoundedSize.

sub lSDCardGetRoundedSize(var long lpvSDCardRoundedSize)

lpvSDRoundedCardSize the rounded size of the card.

The subroutine lSDCardGetRoundedSize calls internally the subroutine lSDCardGetSize.

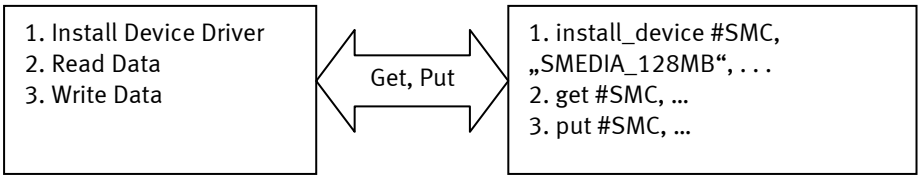
Example for the subroutines lSDCardGetSize and lSDCardGetRoundedSize:

```
long llCardSize, llRoundedCardSize, llCardSizeDiff
. . . . . .
call bSDCardGetSize (llCardSize)
call bSDCardGetRoundedSize (llRoundedCardSize)
llCardSizeDiff = llRoundedCardSize - llCardSize
```

Direct Access to SmartMedia Card

Working with SmartMedia Card using Direct Access

The usual device driver instructions (get, put) are used to read from or write to SmartMedia card directly.



User Function Codes for SmartMedia Device Driver Instructions

The Get and Put instructions are used in combination with user function codes to control SmartMedia card, read and write parameters or data.

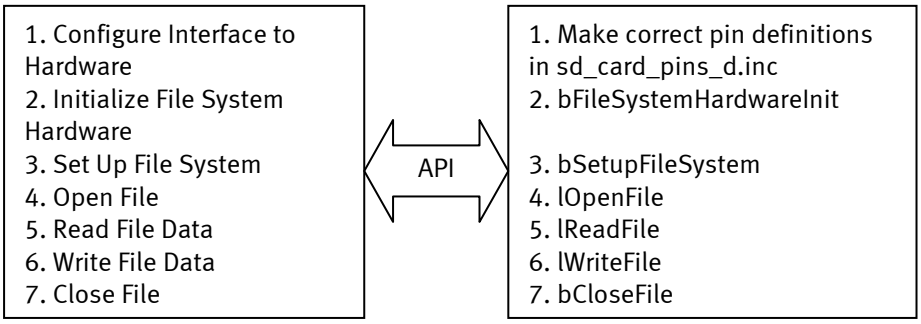
User Function Code	GET	PUT
2 (ufunc_sm_mount)	Mount: ID+Infoset+iBlks	
3 (ufunc_sm_unmount)	UnMount	
4 (ufunc_sm_reset)	Reset Card	
5 (ufunc_sm_getstat)	Read Status Byte	
6 (ufunc_sm_getinfo)	GET ID + Infoset	
7 (ufunc_sm_getiblks)	GET invalid Block Table	
17 (ufunc_sm_eraseblk)	ERASE Block abs (not waiting)	ERASE Block abs (not waiting)
18 (ufunc_sm_rdpage)	READ-1-DATA-Page-abs	
18 (ufunc_sm_wrpage)		Write One Data Page abs.
19 (ufunc_sm_rdpagesp)	Read One Data Page abs. + Spare Field	
19 (ufunc_sm_wrpagesp)		Write One Data Page abs. + Spare Field

24 (ufunc_sm_chk_sfe)	Read & Check Spare Field empty	
25 (ufunc_sm_chk_pae)	Read & Check Page empty	
26 (ufunc_sm_first_eblk)	Find first empty Block	
27 (ufunc_sm_first_epage)	Find first empty Page in Block	

File System (FAT) Support for Memory Cards

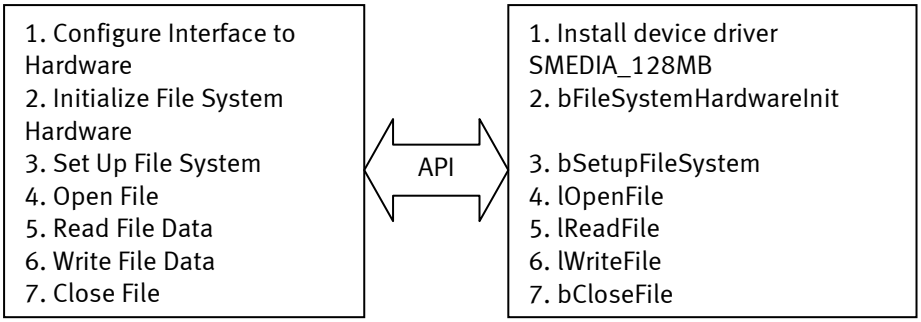
Working with SD/MMC Card using FAT

The following simple steps are necessary to open a file, read from or write to the file and close the file using FAT methods for SD/MMC card.



Working with SmartMedia Card using FAT

The following simple steps are necessary to open a file, read from or write to the file and close the file using FAT methods for SmartMedia card.



Requirements

The Memory Cards (SD/MMC and SmartMedia) must be formatted with FAT 16, to be used with the present FAT routines.

Initialize File System Hardware

sub bFileSystemHardwareInit(var byte bpvHdInitOk)

bpvHdInitOk return value that is set to TRUE on successful initializing and to FALSE on error.

The bFileSystemHardwareInit subroutine calls special subroutines initializing a particular storage medium that is to be used by the file system. This subroutine retrieves also the parameters of the storage medium.

Example for the subroutine bFileSystemHardwareInit:

```
'' name: file_open.tig
. . . . . .
'' initialise file system hardware
call bFileSystemHardwareInit( blHdInitOk )
if blHdInitOk = TRUE then
. . . . . .
endif
```

Set Up File System

sub bSetupFileSystem(var byte bpvIsFSSetupOk)

bpvIsFSSetupOk return value that is set to TRUE on success and to FALSE on error.

The bSetupFileSystem subroutine initializes internal file system data, reads the boot sector and retrieves current file system settings.

Example for the subroutine bSetupFileSystem:

```
'' name: file_open.tig
. . . . . .
'' setup file system
call bSetupFileSystem( blIsFSSetupOk )
if blIsFSSetupOk = TRUE then
. . . . . .
endif
```

Open File

sub lOpenFile(string spFileName$; long lpFlags; var long lpvHandle)

spFileName$ the name of the file that must be opened/created.

lpFlags the open mode flags.

lpvHandle return value that is a new file descriptor for the opened file.

The lOpenFile subroutine creates and returns a new file descriptor for the file named by spFileName$. Initially, the file position indicator for the file is at the beginning of the file.

The lpFlags argument controls how the file is to be opened. This is a bit mask; you create the value by using bitwise OR on the appropriate parameters (using the 'bitor' operator in TB). File status flags lpFlags fall into three following categories.

File Access Modes:

The file access modes allow a file descriptor to be used for reading, writing, or both. The access modes are chosen when the file is opened, and never change.

File Access Mode Constant	Description
O_RDONLY	Open the file for read access
O_WRONLY	Open the file for write access
O_RDWR	Open the file for both reading and writing

O_RDONLY and O_WRONLY are independent bits that can be bitwise-ORed together, and it is valid for either bit to be set or clear. This means that O_RDWR is the same as O_RDONLY bitor O_WRONLY. A file access mode of zero is equal in meaning to O_RDWR.

Open-time Flags:

The open-time flags specify options affecting how open will behave. These options are not preserved once the file is open.

Open-time Flag Constant	Description
O_CREAT	The file will be created if it doesn't already exist
O_EXIST	Check, whether the file exists, don't open the file. In the case of a success the return value is zero, which does not mean that a file descriptor was assigned to an opened file

I/O Operating Modes:

The operating modes affect how input and output operations using a file descriptor work.

I/O Operating Mode Constant	Description
O_APPEND	The bit that enables append mode for the file. If set, then all 'write' operations write the data at the end of the file, extending it, regardless of the current file position. This is the only reliable way to append to a file

The normal return value lpvHandle from lOpenFile is a non-negative long integer file descriptor. In the case of an error, a value of {-1} is returned instead.

```
'' name: file_open.tig
. . . . . .
#define NEW_FILENAME    "Brave_New_File.txt"
. . . . . .
'' create a new file and open it;
'' if the file already exists, just open it;
'' the file path "NEW_FILENAME" is relative to the Current Directory
llFlags = O_RDWR bitor O_CREAT
call lOpenFile( NEW_FILENAME, llFlags, llHandle )
if llHandle <> -1 then
. . . . . .
endif
```

Close File

sub bCloseFile(long lpHandle; var byte bpvIsFileClosed)

lpHandle the file descriptor that was returned by the lOpenFile subroutine.

bpvIsFileClosed return value that is set to FALSE if the file descriptor lpHandle is invalid, otherwise it is TRUE.

The bCloseFile subroutine closes the file descriptor lpHandle.

```
'' name: file_open.tig
. . . . . . . . . . . . . . .
#define NEW_FILENAME    "Brave_New_File.txt"
. . . . . . . . . . . . . . .
'' create a new file and open it;
'' if the file already exists, just open it;
'' the file path "NEW_FILENAME" is relative to the Current Directory
llFlags = O_RDWR bitor O_CREAT
call lOpenFile( NEW_FILENAME, llFlags, llHandle )
if llHandle <> -1 then
  . . . . . . . . . . . . . . .
  '' close the file
  call bCloseFIle( llHandle, blIsFileClosed )
endif
```

Read from File

sub lReadFile(long lpHandle; var string spvBuffer$; long lpSize; var long lpvNumBytesRead)

lpHandle	the file descriptor that was returned by the lOpenFile subroutine.
spvBuffer$	the buffer to store the read data.
lpSize	the maximal size of the data to read.
lpvNumBytesRead	return value: the number of bytes that were actually read, or zero if the end-of-file is reached, or -1 on error.

The lReadFile subroutine reads maximally lpSize bytes from the file with descriptor lpHandle, storing the results to the spvBuffer$ string.

The return value lpvNumBytesRead is the number of bytes actually read. This might be less than lpSize, e.g. if there aren't that many bytes left in the file. Note that reading less than lpSize bytes is not an error.

A value of zero (lpvNumBytesRead = 0) indicates end-of-file (except if the value of the lpSize argument is also zero). This value is not considered to be an error. If you keep calling lReadFile staying at end-of-file, it will keep returning zero and doing nothing else.

If lReadFile returns at least one character, there is no way you can tell whether end-of-file was reached. But if you did reach the end, the next reading will return zero.

In case of an error, lReadFile returns {-1}.

```
'' name: file_open.tig
. . . . . . . . . . . . . . .
#define NEW_FILENAME    "Brave_New_File.txt"
. . . . . . . . . . . . . . .
'' create a new file and open it;
'' if the file already exists, just open it;
'' the file path "NEW_FILENAME" is relative to the Current Directory
llFlags = O_RDWR bitor O_CREAT
call lOpenFile( NEW_FILENAME, llFlags, llHandle )
if llHandle <> -1 then
  slBuffer$ = ""
  llBufSize = 10
  '' read 10 bytes from the file referenced by "llHandle"
  call lReadFile( llHandle, slBuffer$, llBufSize, llNumBytesRead )
  . . . . . . . . . . . . . . . .
  '' close the file
  call bCloseFIle( llHandle, blIsFileClosed )
endif
```

Write to File

sub lWriteFile(long lpHandle; string spBuffer$; long lpSize; var long lpvNumBytesWritten)

lpHandle the file descriptor that was returned by the lOpenFile subroutine.

spBuffer$ the buffer to write the data from.

lpSize the number of bytes of the data to write.

lpvNumBytesWritten return value: the number of bytes that were actually written from the file, or -1 if error occurs.

The lWriteFile subroutine writes lpSize bytes from spBuffer$ to the file with descriptor lpHandle.

The return value is the number of bytes actually written. This may be lpSize, but can be smaller. Your program should call lWriteFile in a loop, iterating until all the data is written.

In the case of an error, lWriteFile returns {-1}.

```
'' name: file_open.tig
. . . . . .
#define NEW_FILENAME    "Brave_New_File.txt"
. . . . . .
'' create a new file and open it;
'' if the file already exists, just open it;
'' the file path "NEW_FILENAME" is relative to the Current Directory
llFlags = O_RDWR bitor O_CREAT
call lOpenFile( NEW_FILENAME, llFlags, llHandle )
if llHandle <> -1 then
  slBuffer$ = "0123456789"
  llBufSize = 10
  '' write 10 bytes to the file referenced by "llHandle"
  call lWriteFile( llHandle, slBuffer$, llBufSize, llNumBytesWritten )
  . . . . . .
  '' close the file
  call bCloseFIle( llHandle, blIsFileClosed )
endif
```

Get File Position

The File Position of a Descriptor specifies the position in the file for the next read or write operation.

sub lGetFilePointer(long lpHandle; var long lpvCurFilePtr)

lpHandle	the file descriptor that was returned by the lOpenFile subroutine.
lpvCurFilePtr	return value: the current file position, measured in bytes from the beginning of the file, or -1 if error occurs.

The lGetFilePointer subroutine is used to read the file position of the file referenced by lpHandle.

The return value lpvCurFilePtr from lGetFilePointer is normally the current file position, measured in bytes from the beginning of the file. If the value of file descriptor is invalid, lGetFilePointer returns a value of {-1}.

```
'' name: file_pointer.tig
. . . . . .
#define NEW_FILENAME    "Brave_New_File.txt"
. . . . . .
call lOpenFile( NEW_FILENAME, llFlags, llHandle )
if llHandle <> -1 then
  . . . . . .
  '' get the current file ponter for the file referenced by "llHandle"
  call lGetFilePointer( llHandle, llCurFilePtr )
  . . . . . .
  '' close the file
  call bCloseFIle( llHandle, blIsFileClosed )
endif
```

Set File Position

sub lSetFilePointer(long lpHandle; long lpOffset; byte bpWhence; var long lpvNewFilePtr)

lpHandle the file descriptor that was returned by the lOpenFile subroutine.

lpOffset the offset to move the file pointer.

bpWhence the position in the file from which the offset should be calculated.

lpvNewFilePtr return value: the resulting file position after moving the file pointer, measured in bytes from the beginning of the file, or -1 if error occurs.

The lSetFilePointer subroutine is used to change the file position of the file referenced by lpHandle.

The bpWhence argument specifies how the lpOffset should be interpreted, and it must be one of the symbolic constants FILE_BEGIN, FILE_CURRENT, or FILE_END.

Offset Direction Constant	Description
FILE_BEGIN	Specifies that lpOffset is a count of characters from the beginning of the file. This count must be positive
FILE_CURRENT	Specifies that lpOffset is a count of characters from the current file position. This count may be positive or negative
FILE_END	Specifies that lpOffset is a count of characters from the end of the file. This count must be positive

The return value lpvNewFilePtr from lSetFilePointer is normally the resulting file position, measured in bytes from the beginning of the file. One can use this return value together with FILE_CURRENT to read the current file position, though the using of lGetFilePointer is more efficient in this case.

If the file position cannot be modified, or the operation is in some way invalid, lSetFilePointer returns a value of {-1}.

The position past the current end can not be set, and the file can not be extended by using of lSetFilePointer.

```
'' name: file_pointer.tig
. . . . . .
#define NEW_FILENAME    "Brave_New_File.txt"
. . . . . .
call lOpenFile( NEW_FILENAME, llFlags, llHandle )
if llHandle <> -1 then
  . . . . . .
  '' move forward the file ponter on 10 positions from the current pos.
  call lSetFilePointer( llHandle, 10, FILE_CURRENT, llNewFilePtrRead )
  . . . . . .
  '' close the file
  call bCloseFIle( llHandle, blIsFileClosed )
endif
```

Get File Size

sub lGetFileSize(long lpHandle; var long lpvFileSize)

lpHandle	the file descriptor that was returned by the lOpenFile subroutine.
lpvFileSize	return value: the file size, measured in bytes, or -1 if error occurs.

The lGetFileSize subroutine is used to read the file size of the file referenced by lpHandle.

The return value lpvFileSize from lGetFileSize is normally the file size, measured in bytes. The subroutine lGetFileSize returns a value of {-1} on error.

```
'' name: file_size.tig
. . . . . .
#define NOT_EMPTY_FILE_NAME     "Not_Empty_File.txt"
. . . . . .
call lOpenFile( NOT_EMPTY_FILE_NAME, llFlags, llHandle )
if llHandle <> -1 then
  . . . . . .
  '' get the file size
  call lGetFileSize( llHandle, llFileSize )
  . . . . . .
  '' close the file
  call bCloseFIle( llHandle, blIsFileClosed )
endif
```

Root Directory, Absolute and Relative Paths

A root directory is a very first directory in a hierarchy. The name of the root directory consists of one character "\" ("/" is also accepted).

An absolute path or full path is a path that points to the same location on the file system regardless of the current working directory or combined paths.
An absolute path begins always with the root directory name.

A relative path is a path relative to the current working directory, so the full absolute path may not need to be given.
A relative path must never have the root directory name as a very first part of the whole path.

Create Directory

sub bCreateDirectory(string spDirName$; var byte bpvIsCreated)

spDirName$ the name of directory to create.

bpvIsCreated return value: TRUE if the creating is successfully completed, FALSE if an error occurs.

The bCreateDirectory subroutine creates a new, empty directory with name spFileName$.

A return value bpvIsCreated of TRUE indicates successful completion, and FALSE indicates failure.

```
'' name: dir_create_del.tig
. . . . . .
#define NEW_DIRNAME     "\New_Dir"
. . . . . .
'' create new directory with the name defined by NEW_DIRNAME
call bCreateDirectory( NEW_DIRNAME, blIsCreated )
if blIsCreated = TRUE then
  . . . . . .
endif
```

Delete File or Directory

sub bDeleteFile(string spFileName$; var byte bpvIsDeleted)

spFileName$ the name of file or directory to delete.

bpvIsDeleted return value: TRUE if the deleting is successfully completed, FALSE if an error occurs.

The bDeleteFile subroutine deletes a file or a directory spFileName$.

A read-only file (i.e. a file with the set “DIR_ATTR_READONLY” attribute) cannot be removed.

A directory must be empty before it can be removed; in other words, it can only contain entries for ‘.’ and ‘..’.

This subroutine returns in bpvIsDeleted TRUE on successful completion, and FALSE on error.

```
'' name: dir_create_del.tig
. . . . . .
#define NEW_DIRNAME     "\New_Dir"
. . . . . .
'' create new directory with the name defined by NEW_DIRNAME
call bCreateDirectory( NEW_DIRNAME, blIsCreated )
if blIsCreated = TRUE then
  . . . . . .
  '' delete the preivously created directory
  call bDeleteFile( NEW_DIRNAME, blIsDirDeleted )
endif
```

Set Current Directory

Current (Working) Directory is a directory to which every not-absolute path is related.

sub bSetCurrentDir(string spNewCurrentDir$;var byte bpvIsDirSet)

spNewCurrentDir$ the name of new current directory.

bpvIsDirSet return value: TRUE if the setting is successfully completed, FALSE if an error occurs.

The bSetCurrentDir subroutine sets Current Directory to the spNewCurrentDir$.

This subroutine returns in bpvIsDirSet TRUE on successful setting, and FALSE on error.

```
'' name: dir_create_del.tig
. . . . . .
#define NEW_DIRNAME     "\New_Dir"
. . . . . .
'' create new directory with the name defined by NEW_DIRNAME
call bCreateDirectory( NEW_DIRNAME, blIsCreated )
if blIsCreated = TRUE then
  . . . . . .
  '' delete the preivously created directory
  call bDeleteFile( NEW_DIRNAME, blIsDirDeleted )
endif
```

File Attributes

File Attribute is a byte value describing the most common properties of any particular file system entry (file or directory). A File Attribute is a combination of following constants:

File Attribute Constant	Description
DIR_ATTR_FILE	The entry is a file
DIR_ATTR_READONLY	The file or directory is read-only. Applications can read the file but cannot write to it or delete it. In the case of a directory, applications cannot delete it
DIR_ATTR_SYSTEM	The file or directory is part of, or is used exclusively by, the operating system
DIR_ATTR_HIDDEN	The file or directory is hidden. It is not included in an ordinary directory listing
DIR_ATTR_VOLUME	Volume label attribute means that this entry contains the disk label in the filename and extension fields. Volume label is valid only in the root directory. Common sense says, there should be only one volume label per disk. For the entry to really contain the volume label, the attribute should be exactly DIR_ATTR_VOLUME
DIR_ATTR_DIRECTORY	The entry is a directory
DIR_ATTR_ARCHIVE	The file or directory is an archive file or directory. Applications use this flag to mark files for backup or removal

Get File Attributes

sub bGetFileAttributes(string spFileName$; var byte bpvFileAttr; var byte bpvAttrReadOk)

spFileName$	the name of file.
bpvFileAttr	return value: the file attributes.
bpvAttrReadOk	return value: TRUE if the reading is successful, FALSE if an error occurs.

The bGetFileAttributes subroutine reads a File Attribute value of the file spFileName$, storing the result in the bpvFileAttr.

This subroutine returns in bpvAttrReadOk TRUE on successful reading, and FALSE on error.

```
'' name: file_attributes.tig
. . . . . .
'' get file attributes
call bGetFileAttributes( NEW_FILENAME, blCurFileAttr, blIsFileAttrOk )
if blIsFileAttrOk = TRUE then
  '' check whether the file is read-only file
  if blCurFileAttr bitand DIR_ATTR_READONLY = DIR_ATTR_READONLY then
    . . . . . .
  endif
endif
```

Set File Attributes

sub bSetFileAttributes(string spFileName$; byte bpNewFileAttr; var byte bpvAttrSetOk)

spFileName$	the name of file.
bpNewFileAttr	the new file attributes.
bpvAttrSetOk	return value: TRUE if the writing is successful, FALSE if an error occurs.

The bSetFileAttributes subroutine writes the new File Attribute value bpNewFileAttr of the file spFileName$.

This subroutine returns in bpvAttrSetOk TRUE on successful writing, and FALSE on error.

```
'' name: file_attributes.tig
. . . . . .
'' make the file to the read-only one setting an appropriate attribute
blNewFileAttr = blNewFileAttr bitor DIR_ATTR_READONLY
call bSetFileAttributes( NEW_FILENAME, blNewFileAttr, blIsFileAttrOk )
if blIsFileAttrOk = TRUE then
    . . . . . .
endif
```

File Time and Date

Time and Date stamps are represented in the FAT system in the following special formats.

The file time format:

Bits	Range	Translated Range	Valid Range	Description
0..4	0..31	0..62	0..59	Seconds/2
5..10	0..63	0..63	0..59	Minutes
11..15	0..31	0..31	0..23	Hours

The file date format:

Bits	Range	Translated Range	Valid Range	Description
0..4	0..31	0..31	1..28 up to 1..31	Day
5..8	0..15	0..15	1..12	Month
9..15	0..127	1980..2107	1980..2107	Year (+1980)

Convert Time and Date Stamps

The FAT time and date stamps can be converted from packed representation to the usual second, minute, hour and day, month, year values and vice versa by using of the subroutines lUnPackTime, lUnPackDate, wPackTime, and wPackDate.

sub lUnPackTime(word wpPackedTime; var long lpvSec, lpvMin, lpvHour)

wpPackedTime	the time packed in FAT format.
lpvSec	return value: unpacked seconds.
lpvMin	return value: unpacked minutes.
lpvHour	return value: unpacked hours.

sub lUnPackDate(word wpPackedDate; var long lpvDay, lpvMonth, lpvYear)

wpPackedDate	the date packed in FAT format.
lpvDay	return value: unpacked day.
lpvMonth	return value: unpacked month.
lpvYear	return value: unpacked year.

sub wPackTime(long lpSec, lpMin, lpHour; var word wpvPackedTime)

lpSec	unpacked seconds.
lpMin	unpacked minutes.
lpHour	unpacked hours.
wpPackedTime	return value: the time packed in FAT format.

sub wPackDate(long lpDay, lpMonth, lpYear; var word wpvPackedDate)

lpDay	unpacked day.
lpMonth	unpacked month.
lpYear	unpacked year.

wpvPackedDate return value: the date packed in FAT format.

```
'' name: file_time.tig
. . . . . .
'' set the following new file time;
'' at first, pack it
call wPackDate( 14, 7, 2002, wlCreateDate )
call wPackTime( 30, 59, 23, wlCreateTime )
call wPackDate( 14, 7, 2002, wlAccessDate )
call wPackDate( 14, 7, 2002, wlWriteDate )
call wPackTime( 30, 59, 23, wlWriteTime )
call bSetFileTime( NEW_FILENAME, wlCreateDate, wlCreateTime, wlAccessDate, &
        wlWriteDate, wlWriteTime, blIsTimeOk )
call vSynchronizeFS()

'' get the actualy set file time
call bGetFileTime( NEW_FILENAME, wlCreateDate, wlCreateTime, wlAccessDate, &
        wlWriteDate, wlWriteTime, blIsTimeOk )
'' unpack all date and time entries
call lUnPackDate( wlCreateDate, llDay, llMon, llYear )
call lUnPackTime( wlCreateTime, llSec, llMin, llHour )
call lUnPackDate( wlAccessDate, llDay, llMon, llYear )
call lUnPackDate( wlWriteDate, llDay, llMon, llYear )
call lUnPackTime( wlWriteTime, llSec, llMin, llHour )
```

Get File Time

sub bGetFileTime(string spFileName$; var word wpvCreateDate, wpvCreateTime, wpvAccessDate, wpvWriteDate, wpvWriteTime; var byte bpvIsTimeRead)

spFileName$	the name of file.
wpvCreateDate	the date the file was created.
wpvCreateTime	the time the file was created.
wpvAccessDate	the date the file was last accessed.
wpvWriteDate	the date the file was last modified.
wpvWriteTime	the time the file was last modified.
bpvIsTimeRead	return value: TRUE if the reading is successful, FALSE if an error occurs.

The bGetFileTime subroutine retrieves the date and time that a file spFileName$ was created, last accessed, and last modified.

All the time and date fields are represented in the format described in the "File Time and Date ".

```
'' name: file_time.tig
. . . . . .
'' get the actualy set file time
call bGetFileTime( NEW_FILENAME, wlCreateDate, wlCreateTime, wlAccessDate, &
        wlWriteDate, wlWriteTime, blIsTimeOk )
'' unpack all date and time entries
call lUnPackDate( wlCreateDate, llDay, llMon, llYear )
call lUnPackTime( wlCreateTime, llSec, llMin, llHour )
call lUnPackDate( wlAccessDate, llDay, llMon, llYear )
call lUnPackDate( wlWriteDate, llDay, llMon, llYear )
call lUnPackTime( wlWriteTime, llSec, llMin, llHour )
```

Set File Time

sub bSetFileTime(string spFileName$; word wpCreateDate, wpCreateTime, wpAccessDate, wpWriteDate, wpWriteTime; var byte bpvIsTimeWritten)

spFileName$	the name of file.
wpCreateDate	the date the file was created.
wpCreateTime	the time the file was created.
wpAccessDate	the date the file was last accessed.
wpWriteDate	the date the file was last modified.
wpWriteTime	the time the file was last modified.
bpvIsTimeWritten	return value: TRUE if the writing is successful, FALSE if an error occurs.

The bSetFileTime subroutine sets the date and time that a file spFileName$ was created, last accessed, and last modified.

All the time and date fields are represented in the format described in the "File Time and Date".

```
'' name: file_time.tig
. . . . . .
'' set the following new file time;
'' at first, pack it
call wPackDate( 14, 7, 2002, wlCreateDate )
call wPackTime( 30, 59, 23, wlCreateTime )
call wPackDate( 14, 7, 2002, wlAccessDate )
call wPackDate( 14, 7, 2002, wlWriteDate )
call wPackTime( 30, 59, 23, wlWriteTime )
call bSetFileTime( NEW_FILENAME, wlCreateDate, wlCreateTime, wlAccessDate, &
        wlWriteDate, wlWriteTime, blIsTimeOk )
```

Find File

Two subroutines described below return the result of the searching in a string used as a memory block storing the data of different types and sizes. The particular fields of such a block can be accessed by means of the built-in functions (like nfroms, rfroms, mid$ etc.) reading the definite number of bytes from the specific offset into a variable. The following offset and size values can be applied for accessing the information about a found file:

Offset	Size	Description
FFD_ATTR_OFFS	FFD_ATTR_SIZE	file attribute
FFD_CREATE_TIME_MS_OFFS	FFD_CREATE_TIME_MS_SIZE	ms part of file creating time
FFD_CREATE_TIME_OFFS	FFD_CREATE_TIME_SIZE	file creating time
FFD_CREATE_DATE_OFFS	FFD_CREATE_DATE_SIZE	file creating date
FFD_ACCESS_DATE_OFFS	FFD_ACCESS_DATE_SIZE	date of the last file access
FFD_SIZE_OFFS	FFD_SIZE_SIZE	file size
FFD_NAME_OFFS	FFD_NAME_SIZE	file name (max. 8 symbols)
FFD_EXT_OFFS	FFD_EXT_SIZE	file extension (max. 3 symbols)
FFD_LONG_NAME_OFFS	FFD_LONG_NAME_SIZE	long file name
FFD_ABRIDGED_NAME_OFFS	FFD_ABRIDGED_NAME_SIZE	abridged file name

Note:

1. The following subroutines searches only for short file names (the names in the format 8.3). So two long names with 6 or more equal first characters can not be differentiated.
2. If the file name was found and there is an entry for the long name, this long name will be saved in the memory block at the FFD_LONG_NAME_OFFS offset or at the FFD_ABRIDGED_NAME_OFFS offset (if this form of presentation was preferred).
3. The file name at the FFD_NAME_OFFS offset is extended with blanks up to FFD_NAME_SIZE (8) size; the file extension at the FFD_EXT_OFFS offset – up to FFD_EXT_SIZE (3) size.
4. The abridged form of presentation makes sense if one knows that the file name is in the format 8.3 and one would like to use the found name (placed at the FFD_ABRIDGED_NAME_OFFS offset in the format 8.3 with dot and without extending blanks) directly in the next file operation.

5. The size of the memory block can be equal or greater than FFD_STRUCT_SHORT_SIZE.
6. The following size constants are predefined:
   - FFD_STRUCT_SHORT_SIZE – without fields for the long or abridged file name
   - FFD_STRUCT_ABRIDGED_SIZE - FFD_STRUCT_SHORT_SIZE + the maximal length of the file name in the abridged form (FFD_NAME_SIZE + FFD_EXT_SIZE + 1[for "dot"])
   - FFD_STRUCT_FULL_SIZE - FFD_STRUCT_SHORT_SIZE + the maximal length of the long file name
   - FFD_STRUCT_DEFAULT_SIZE - FFD_STRUCT_ABRIDGED_SIZE

Search for File Name

sub bFindFirstFile(string spSearchedFileName$; var string spvFfdStruct$; var byte bpvFound)

spSearchedFileName$ the name of the file to search for.

spvFfdStruct$ return value: data block (string) that contains information about the file if it was found.

bpvFound return value: TRUE if the file is found, FALSE if an error occurs.

The bFindFirstFile subroutine searches a directory for a file whose name matches the specified spSearchedFileName$ filename
and fills on success the spvFfdStruct$ string with the information about the found file. The spSearchedFileName$ filename can contain wildcard characters (* and ?).

sub bFindNextFile(var string spvFfdStruct$; var byte bpvFound)

spvFfdStruct$ return value: data block (string) that contains information about the file if it was found.

bpvFound return value: TRUE if the file is found, FALSE if an error occurs.

The bFindNextFile subroutine continues the searching a directory for a file whose name matches the filename that was specified in the previous call of the bFindFirstFile subroutine in the parameter spSearchedFileName$ and fills on success the spvFfdStruct$ string with the information about the found file. The process begins at the position next to the position where the previous search was successfully completed by the bFindFirstFile or bFindNextFile subroutine.

```
'' name: file_time.tig
. . . . . .
'' look for FILENAME
call bFindFirstFile(FILENAME, slFfdStruct$, blFound )
'' read the attributes and exact name for FILENAME
blFileAttr = nfroms( slFfdStruct$, FFD_ATTR_OFFS, FFD_ATTR_SIZE )
slFileName$ = mid$( slFfdStruct$, FFD_LONG_NAME_OFFS, len(slFfdStruct$) -
FFD_LONG_NAME_OFFS )
'' look for FILENAME again
call bFindNextFile( slFfdStruct$, blFound )
. . . . . .
```

Get Information About File System

sub bGetFileSystemInfo(var string spvBootRecord$; var byte bpvIsBootRecRead)

spvBootRecord$ return value: data block (string) that contains information about the the file system.

bpvIsBootRecRead return value: TRUE if the boot record is successfully read, FALSE if an error occurs.

The bGetFileSystemInfo subroutine reads the information about the file system into the spvBootRecord$ string. The information is extracted from the boot record of a FAT-formatted storage media.

The bGetFileSystemInfo subroutine saves the result in the spvBootRecord$ string used as a memory block storing the data of different types and sizes. The particular fields of such a block can be accessed by means of the built-in functions (like nfroms, rfroms, mid$ etc) reading the definite number of bytes from the specific offset into a variable. The following offset and size values can be applied for accessing the information about a the storage media:

Offset	Size	Description
BS_OEM_NAME_POS	BS_OEM_NAME_SIZE	the system that formatted the disk
BPB_BYTES_PER_SECT_POS	BPB_BYTES_PER_SECT_SIZE	the length in bytes of one physical sector
BPB_SECT_PER_CLUSTER_POS	BPB_SECT_PER_CLUSTER_SIZE	the number of sectors in one logical cluster
BPB_RESERVED_SECT_POS	BPB_RESERVED_SECT_SIZE	the number of reserved sectors
BPB_NUMBER_OF_FATS_POS	BPB_NUMBER_OF_FATS_SIZE	the number of File Allocation Tables

BPB_ROOT_ENTRIES_POS	BPB_ROOT_ENTRIES_SIZE	the number of entries in the root directory
BPB_TOTAL_SECT_POS	BPB_TOTAL_SECT_SIZE	total number of sectors on the disk
BPB_MEDIA_POS	BPB_MEDIA_SIZE	media descriptor
BPB_SECT_PER_FAT_POS	BPB_SECT_PER_FAT_SIZE	the number of sectors in one FAT
BPB_HIDDEN_SECT_POS	BPB_HIDDEN_SECT_SIZE	the number of hidden sectors
BPB_TOTAL_SECT_BIG_POS	BPB_TOTAL_SECT_BIG_SIZE	the a number of sectors if greater 65535
BS_VOLUME_LABEL_POS	BS_VOLUME_LABEL_SIZE	the disk label
BS_FILE_SYSTEM_POS	BS_FILE_SYSTEM_SIZE	the file system name (FAT12/16)

Note:

The size of the spvBootRecord$ string must be equal or greater than BOOT_RECORD_SIZE.

Synchronize the File System

For reasons of efficiency, some intensively used data structures of the FAT file system are temporary stored in the RAM memory while the file system operations are performed. Before the permanent storage media (e.g. SD-Card) is unplugged, all the data structures must be copied from the RAM to the permanent storage media. The process of copying of the data is named “synchronization”. The synchronization may be performed either by calling the vSynchronizeFS subroutine explicitly or by implementing a task, that sets a value of the synchronization timeout using the lSetSyncTimeout subroutine and calls in the endless loop the bSynchronizeFSRegularly subroutine.
The synchronization timeout values are measured in seconds.

sub vSynchronizeFS()

The vSynchronizeFS subroutine writes to the media all data structures that were temporary saved in the RAM.

sub bGetSyncTimeout(var long lpvSyncTimeout; var long lpvCurSyncTimeoutCounter)

lpvSyncTimeout return value: the recently set synchronization timeout.

lpvSyncTimeoutCounter return value: the current value of the timeout counter.

The lGetSyncTimeout subroutine returns the recently set synchronization timeout value in the lpvSyncTimeout and the current value of the timeout counter in the lpvCurSyncTimeoutCounter.

If the timeout values have not been yet initialised, the lGetSyncTimeout subroutine returns –1 in the both lpvSyncTimeout and lpvCurSyncTimeoutCounter.

sub bSetSyncTimeout(long lpNewSyncTimeout; var long lpvPrevSyncTimeout)

lpNewSyncTimeout the new value of the synchronization timeout.

lpvPrevSyncTimeout return value: the previous value of the synchronization timeout.

The lSetSyncTimeout subroutine sets the new synchronization timeout value to the lNewSyncTimeout value.

The lSetSyncTimeout subroutine returns the previously set synchronization timeout value in the lpvPrevSyncTimeout or –1 if it has not been yet initialised.

sub bSynchronizeFSRegularly(var byte bpvTimeoutReached)

bpvTimeoutReached return value: TRUE if the synchronisation was performed, else FALSE.

The bSynchronizeFSRegularly subroutine calls the vSynchronizeFS subroutine when the synchronization timeout is over.