Basic Tiger File System for SmartMedia

Version 1.04

Introduction

Basic Tiger File System (BTFS) is a collection of subroutines written in the Tiger Basic programming language and implementing general functionality of FAT file system for permanent storage devices. BTFS consists of three hierarchical layers: File System API, FAT implementation, special hardware support. A device driver for the particular hardware underlies the BTFS.

BTFS for SmartMedia Card

BTFS for SmartMedia Card has the following structure:
FS API ←FAT ←SmartMedia Routines ←SmartMedia Device Driver.

FS API is a set of subroutines working with files and directories. FS API layer is considered to be the most interesting layer for an application programmer and exactly this layer is described more detailed in the present document.

FAT is an implementation of FAT12/FAT16 file system (with long names support).

SmartMedia Routines is a set of subroutines written with regard to the specifications of SmartMedia card devices.

SmartMedia Device Driver is a Basic Tiger device driver implementing elementary interface between SmartMedia hardware and a Tiger Basic application.

BTFS for SmartMedia File List

FS Include Files (directory "File_System")

fs_conf.inc - definitions that can be changed by user
fs_coinc.inc - definitions relevant for all FS layers; co-including of all FS components. *Only this file must be explicitly included in the Tiger Basic application using BTFS*.
fs_inx_i.inc - implementation of FS API and some maintaining subroutines.
fs_inx_d.inc - definitions relevant for FS API.
fs_fat_i.inc - implementation of FAT12/FAT16 with longnames support.
fs_fat_d.inc - definitions useful for FAT12/FAT16 implementation.
fs_fmt_i.inc - implementation of formatting process.
fs_dat_i.inc - implementation of date and time conversions.
fs_dat_d.inc - definitions relevant for date and time conversions.
fs_hal_d.inc - definition of Hardware Abstraction Layer; HAL is used to simplify the adaptation of the file system subroutines to the working with other storage devices.
fs_smc_i.inc - implementation of subroutines working with SmartMedia and conforming to the SmartMedia specifications and to the special features of the SmartMedia device driver.

fs_smc_d.inc - definitions relevant for SmartMedia subroutines.
fs_ecc_i.inc - implementation of ECC calculation for SmartMedia.

FS Examples (directory "File_System")

dir_create_del.tig, file_open.tig, file_size.tig, file_pointer.tig, file_attributes.tig, file_time.tig, file_format.tig, file_sync.tig, file_copy.tig, file_find.tig, get_hd_info.tig, get_fs_info.tig

SmartMedia Device Drivers (directory "TB_Drivers" or "Bin")

smedia_16mb.tdd, smedia_32mb.tdd, smedia_64mb.tdd, smedia_128mb.tdd
Any device driver fits for all SmartMedia cards of the exact or smaller size.

SmartMedia Functions (directory "TB_System_Files" or "Bin")

Some new built-in functions are extensively used by the BTFS subroutines. The functions are located in the following enclosed system files:
tac0000.tac, tac0000_.tac, tac0100.tac, tac0100_.tac
The enclosed system files require the Tiger Basic compiler version 5.01 or higher.

SmartMedia Low Level Examples (directory "Random_Access")

smedia_test_era_wr_rd_ser0_v03.tig,
smedia_hex_dump_to_ser_02.tig
Note: This test may destroy very important SmartMedia header information and make the SmartMedia card unusable.

Supported SmartMedia Card Types and Other Limitations

The following SmartMedia card types are supported by BTFS at present: 1Mb, 2Mb, 4Mb, 8Mb, 16Mb, 32Mb, 64Mb, 128Mb.

Most formatting programs use FAT12/FAT16 format for the various types of SmartMedia cards, but can be set to use other formats. You should avoid this as only FAT12/FAT16 is supported by BTFS.

Although long file names are supported, it's not possible to differentiate files with identical first 6 characters.

The BTFS subroutines are not re-entrant. Be careful using the BTFS subroutines in the different tasks.

BTFS System Requirements

BTFS requires the Tiger Basic compiler version 5.01 or higher. *The enclosed system files (extension: TAC) must be copied to the "..\Bin" directory of the Tiger Basic software.*

File System API (application program interface)

File System Setup

Initialising the File System Hardware

Subroutine:
sub bFileSystemHardwareInit(var byte bpvHdInitOk)

The *bFileSystemHardwareInit* subroutine calls special subroutines initializing a particular storage medium (f.e.: SmartMedia) that is to be used by the file system. This subroutine retrieves also the parameters of the storage medium.

This subroutine returns in *bpvHdInitOk* TRUE on successful initializing, and FALSE on error.

Be prepared: This subroutine may take a long time when run with SmartMedia.

Example: all

Setting Up the File System

Subroutine:
sub bSetupFileSystem(var byte bpvIsFSSetupOk)

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

This subroutine returns in *bpvIsFSSetupOk* TRUE on success, and FALSE on error.

Example: nearly all

Opening and Closing Files

Opening the File

Subroutine:
<u>sub lOpenFile(string spFileName$; long lpFlags; var long lpvHandle)</u>

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.

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|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.

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.

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.

Example: “file_open.tig”

Closing the File

Subroutine:
<u>sub bCloseFile(long lpHandle; var byte bpvIsFileClosed)</u>

The *bCloseFile* subroutine closes the file descriptor *lpHandle*.

The normal return value *bpvIsFileClosed* from *bCloseFile* is TRUE. If the file descriptor *lpHandle* is invalid, the value *bpvIsFileClosed* is assigned to FALSE.

Example: “file_open.tig”

File Input and File Output

Reading the File

Subroutine:
<u>sub lReadFile(long lpHandle; var string spvBuffer$; long lpSize; var long lpvNumBytesRead)</u>

The *lReadFile* subroutine reads up to *lpSize* bytes from the file with descriptor *lpHandle*, storing the results in the *spvBuffer$*. (This is not necessarily a character string, and no terminating null character is added.)

The return value *lpvNumBytesRead* is the number of bytes actually read. This might be less than *lpSize*; for example, 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 indicates end-of-file (except if the value of the *lpSize* argument is also zero). This is not considered an error. If you keep calling *lReadFile* while 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 read will return zero.
In case of an error, *lReadFile* returns {-1}.

Example: “file_open.tig”

Writing the File

Subroutine:
<u>sub lWriteFile(long lpHandle; string spBuffer$; long lpSize; var long lpvNumBytesWritten)</u>

The *lWriteFile* subroutine writes up to *lpSize* bytes from *spBuffer$* to the file with descriptor *lpHandle*. The data in *spBuffer$* is not necessarily a character string and a null character is output like any other character.

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}.

Example: “file_open.tig”

Setting and Getting the File Position of a Descriptor

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

Getting the File Position

Subroutine:
sub lGetFilePointer(long lpHandle; var long lpvCurFilePtr)

The *lGetFilePointer* subroutine is used to read the file position of the file with descriptor *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}.

Example: “file_pointer.tig”

Setting the File Position

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

The *lSetFilePointer* subroutine is used to change the file position of the file with descriptor *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.

FILE_BEGIN
: Specifies that bpWhence is a count of characters from the beginning of the file. This count must be positive.

FILE_CURRENT
: Specifies that bpWhence is a count of characters from the current file position. This count may be positive or negative.

FILE_END
: Specifies that bpWhence 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. You can use this feature together with FILE_CURRENT to read the current file position, though the using of *lGetFilePointer* is more efficient.
If the file position cannot be changed, 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*.

Example: “file_pointer.tig”

Getting the File Size

Subroutine:
sub lGetFileSize(long lpHandle; var long lpvFileSize)

The *lGetFileSize* subroutine is used to read the file size of the file with descriptor *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.

Example: “file_size.tig”

Creating Directories

Subroutine:
sub bCreateDirectory(string spFileName$; var byte bpvIsCreated)

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

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

Example: “dir_create_del.tig”

Deleting Files and Directories

Subroutine:
sub bDeleteFile(string spFileName$; var byte bpvIsDeleted)

The *bDeleteFile* subroutine deletes the file or the 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.

Example: “dir_create_del.tig”

Setting Current Directory

Current Directory is a directory to which every not absolute path is related. A root directory name consists of one character "\" ("/" is also accepted). An absolute path begins always with the root directory name. A relative path must never have the root directory name as a very first part of the whole path.

Subroutine:
sub bSetCurrentDir(string spNewCurrentDir$;var byte bpvIsDirSet)

The *bSetCurrentDir* subroutine sets Current Directory to the *spNewCurrentDir$*.

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

Example: “dir_create_del.tig”

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:

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.

Getting the File Attributes

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

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.

Example: “file_attributes.tig”

Setting the File Attributes

Subroutine:
<u>sub bSetFileAttributes(string spFileName$; byte bpNewFileAttr; var byte bpvAttrSetOk)</u>

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

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

Example: “file_attributes.tig”

File Time

Time and Date Format

The file time fields have the following 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 fields have the following 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, add 1980 to convert

Getting the File Time

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

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

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.

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

Example: “file_time.tig”

Setting the File Time

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

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

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.

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

Example: “file_time.tig”

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

Searching for the file name

Subroutine:
<u>sub bFindFirstFile(string spSearchedFileName$; var string spvFfdStruct$; var byte bpvFound)</u>

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 ?).

This subroutine returns in *bpvFound* TRUE on success, and FALSE on error.

Subroutine:
<u>sub bFindNextFile(var string spvFfdStruct$; var byte bpvFound)</u>

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.

This subroutine returns in *bpvFound* TRUE on success, and FALSE on error.

Example: “file_find.tig”

Getting the information about the storage media

Subroutine:
sub bGetHardwareInfo(var string spvInfoSet$; var byte bpvIsRead)

The *bGetHardwareInfo* subroutine reads the information about the currently used storage media into the *spvInfoSet$* string.

The *bGetHardwareInfo* subroutine returns TRUE in the *bpvIsRead* on successful reading, and FALSE on error.

The *bGetHardwareInfo* subroutine saves the result in the *spvInfoSet$* string used as a memory block storing the data of different types and sizes. The particular fields of such a block can be accessed by means of the built-in functions (like nfroms, rfroms, mid$ 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
HDI_MAKER_CODE_POS	HDI_MAKER_CODE_SIZE	Manufacturer Code
HDI_ID_CODE_POS	HDI_ID_CODE_SIZE	Card Identifier
HDI_BYTES_IN_SPARE_POS	HDI_BYTES_IN_SPARE_SIZE	Number of Bytes in Spare Field
HDI_BYTES_IN_PAGE_POS	HDI_BYTES_IN_PAGE_SIZE	Number of DATA Bytes in a Page
HDI_PAGES_IN_BLOCK_POS	HDI_PAGES_IN_BLOCK_SIZE	Number of Pages in a Block
HDI_BYTES_IN_BLOCK_POS	HDI_BYTES_IN_BLOCK_SIZE	Number of DATA Bytes in a Block
HDI_NO_OF_BLOCKS_POS	HDI_NO_OF_BLOCKS_SIZE	Total Number of Blocks
HDI_ADR_HIGH_BLOCK_POS	HDI_ADR_HIGH_BLOCK_SIZE	Base Address of the highest Block
HDI_ADR_END_POS	HDI_ADR_END_SIZE	End Address = First Address after the last Byte

Note:
The size of the *spvInfoSet$* string must be equal or greater than HDI_BLOCK_SIZE.

Example: “get_hd_info.tig”

Getting the information about the file system

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

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 returns TRUE in the *bpvIsBootRecRead* on successful reading, and FALSE on error.

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.

Example: “get_fs_info.tig”

Formatting the Storage Media

Subroutine:
sub bFormatMediaLogicalWin(var byte bpvSuccess)

The *bFormatMediaLogicalWin* subroutine formats a storage media (f.e. SmartMedia) using the settings preferred by the Windows own formatting routines.

This subroutine returns in *bpvSuccess* TRUE on success, and FALSE on error.

Subroutine:
sub bFormatMediaLogical(var byte bpvSuccess)

The *bFormatMediaLogical* subroutine formats a storage media (f.e. SmartMedia) using the settings recommended by the SSFDC Forum.

This subroutine returns in *bpvSuccess* TRUE on success, and FALSE on error.

Example: “file_format.tig”

Synchronizing 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 (f. e. SmartMedia) 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.

Subroutine:
<u>sub vSynchronizeFS()</u>

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

Subroutine:
<u>sub lGetSyncTimeout(var long lpvSyncTimeout; var long lpvCurSyncTimeoutCounter)</u>

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.*

Subroutine:
<u>sub lSetSyncTimeout(long lNewSyncTimeout; var long lpvPrevSyncTimeout)</u>

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.

Subroutine:
<u>sub bSynchronizeFSRegularly(var byte bpvTimeoutReached)</u>

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

This subroutine returns in the *bpvTimeoutReached* TRUE if the synchronisation was performed, else FALSE is returned.

Example: “file_sync.tig”

What Must Be Done

1. Some subroutines are too slow. The execution speed must be increased by means of improved algorithms or built-in functions written in the processor language directly.

2. ECC correction process for SmartMedia is not implemented at the moment.

3. Although long file names are supported, it’s not possible to differentiate files with identical first 6 characters.

4. The information about errors is very scanty. The error messages must be extended. Probably, something like the GetLastError subroutine will be implemented.

5. The subroutines were tested with 8Mb, 32Mb, 64Mb SmartMedia cards. Additional tests would be useful.

6. It is conceivable to use the BTFS with other kinds of storage media, not only with SmartMedia card. For example, one can implement the hardware support layer for the Basic Tiger internal user flash.

7. The BTFS subroutines are not re-entrant. It can be important to find a way to make the BTFS subroutines re-entrant without compromising the efficiency.

8. More comments in the programs and better documentation is everyone’s most fervent wish.☺

Useful References

1. SmartMedia Card Specifications:

   http://www.ssfdc.or.jp/english/index.htm

2. About FAT:

   http://averstak.tripod.com/fatdox/00dindex.htm

   http://msdn.microsoft.com/