The Fat32 Library (Version 3.0)

2014-12-27

1 Content

2 Library Overview

There are in total 4 Fat32 libraries:

- **Fat32_1**: only one file in one directory on one Fat32 device
- **Fat32_2**: more than one file in more than one directory on one Fat32 Device
- **Fat32_1_MD**: only one file in one directory on one or more Fat32 devices
- **Fat32_2_MD**: more than one file in more than one directory on one or more Fat32 Devices

3 Overview of procedures and functions

4 About this document

This document explains the functionality of the **Fat32_2** library. It can also be used for the **Fat32_1** library with the differences that:

- There can only be one directory selected and one file open at any time in **Fat32_1**.
- The first parameter (of type "TFileVar") in most routines should be omitted when using **Fat32_1**.

The **Fat32_x_MD** particularities are described in section 12.

5 Overview

This libraries **Fat32_1, Fat32_1_MD, Fat32_2** and **Fat32_2_MD** enable you to handle **SD/MMC/CF cards and IDE hard disks** formatted in Fat32. They can handle **subdirectories** and **long filenames**. At this moment they can only be used with the **P18Fxxx PIC range** and the **PIC24F** range, the stack is too deep to run it on a P12 or P16 PIC. They use as less as possible ram: the same sectorbuffer is used for the different Fat32 activities. In the Fat32_x_MD libraries every "device" has its own sectorbuffer.

The 4 libraries only support media with a sectorsize of 512 bytes.
The maximum **filesize** is 4GB, the medium can have more capacity (limited by the Fat32 specification).

5.1 Features

- FAT32 File System
- Any number of files/directories can be accessed concurrently[1]
- More than one Fat32 device can handled simlutaneously[2]
- Subdirectory's
- Long Filenames
- Random file access ("Seek" procedure available)
- No "typed" files: all files are of type "file of byte" (or "file of char" if you want).
- Both byte and buffer read/write routines, string write routines
- FindFirst and FindNext routines, with or without filename
- Rename routine, both for files and directories
- Deletion of files and directories (recursive)
- Flush routine
- Swap file for direct (fast) sector access
- Raw file (logical) sector read and write
- Defragment and clean the current directory

[1] Only for the Fat32_2 and Fat32_2_MD libraries, not for the Fat32_1 and Fat32_1_MD libraries, depending on the available ram memory
[2] Only for the Fat32_x_**MD** libraries.

- Count number of files in the current directory of which the name is in a set of names and the attributes comply.
- Create files containing the content of the current directory, both in text and html format.
- Giving the total, free and used card space in byte, Kilobytes, Megabytes (rounded down) and Gigabytes (rounded down).
- Return the Volume Label.
- Make a copy of a file.

5.2 Particularities

- Medium independent, the using software must define 2 functions to read and write sectors from the hardware and (optionally) one function to obtain the medium size
- All file/directory manipulations take place in the current directory (e.g. file/dir searching, deleting, creation, ...)
- **Directories** are manipulated with **specific directory commands** (like "Fat32_MkDir", "Fat32_RmDir", etc...), **not** with the standard file commands (like "Fat32_assign", "Fat32_delete", etc...).
- The routines "Fat32_FindFirst" and "Fat32_"FindNext" can however be used to find any type of file (so also directories, VolumeId's etc.), again only in the current directory.
  An other exception is "Fat32_Rename", which is used both for files and directories.
- "Fat32_Flush" permits to write a (not yet actually written) sectorbuffer and the current filesize to the card/disk . When using this routine after each write action, there is no need any more to close and re-open again after each write action to ensure data safety. After a "flush" the current file is still open (assigned).
- Most non file related routines (like the directory related ones) close the currently open file.

6 General

This library can handle only media formatted in the FAT32 format, with or without master boot record (containing partitioning info).

For **Fat32_2** and **Fat32_2_MD** only:

Are capable of having more than one file open at a time[3].
The way to achieve the "multi" file capability is to have a special variable for each path (directory + file) that must be handled. This variable(s) is of type "**TFileVar**".

So, for each file to be opened (simultaneously) one has to define such a variable, e.g.:

```
var File1, File2, File3, File4: TFileVar;
```

In the example above 4 files can be opened simultaneously.

Each of those variables can point to another directory and to a file in that directory. Most of the functions available in this library have as their first parameter a variable of type "TFileVar"[4].

Variables of type "TFileVar" will be called *File variables* in the rest of this document.

[3] Only for the Fat32_2 library , not for the Fat32_1 library
[4] Only for the Fat32_2 library, not for the Fat32 library: no "TFileVar" variables.

For **Fat32_1_MD** and **Fat32_2_MD**:

Are capable of handling more than one Fat32 Device simultaneously (e.g. an SDMMC and a USB stick). See section 12 for the Fat32_x_MD particularities.

7 The media hardware drivers

Before all described below can be put to work the *user of the library has to provide the routines to read and write one sector* from the medium that contains the Fat32 system and files. Additionally, if **Fat32_Format** is used, the user has to provide the "**Fat32_Dev_Capacity_Sectors**" function.

The signature of the two first procedures is:

`function Fat32_Dev_Read_Sector(Sector: DWord; var Buffer: array[512] of byte): boolean;`
Returns true when successful

and

`function Fat32_Dev_Write_Sector(Sector: DWord; var Buffer: array[512] of byte): boolean;`
Returns true when successful

See section 12.1.2 for the media drivers for the Fat32_x_MD libraries.

An example of these could be e.g.:

```
{$DEFINE MMC}

function Fat32_Dev_Read_Sector(Sector: DWord; var Buffer: array[512] of byte): boolean;
// returns true when successful
var Tmp: byte;
begin
{$IFDEF MMC}
  Tmp := Mmc_Read_Sector(Sector, Buffer);
  Result := (Tmp = 0);
{$ENDIF}

{$IFDEF SDMMC_SPI1}
  Result := SDMMC_ReadSector(Sector, Buffer);
{$ENDIF}

{$IFDEF SDMMC_SPI2}
  Result := SDMMC_ReadSector(Sector, Buffer);
{$ENDIF}

{$IFDEF CF}
  CF_Read_Sector(Sector, Buffer);
  Result := true;
{$ENDIF}

{$IFDEF IDE}
  Result := IDE_ReadSector(Sector, Buffer);
{$ENDIF}
end;

function Fat32_Dev_Write_Sector(Sector: DWord; var Buffer: array[512] of byte): boolean;
// returns true when successful
var Tmp: byte;
```

```
begin
{$IFDEF MMC}
  Tmp := Mmc_Write_Sector(Sector, Buffer);
  Result := (Tmp = 0);
{$ENDIF}

{$IFDEF SDMMC_SPI1}
  Result := SDMMC_WriteSector(Sector, Buffer);
{$ENDIF}

{$IFDEF SDMMC_SPI2}
  Result := SDMMC_WriteSector(Sector, Buffer);
{$ENDIF}

{$IFDEF CF}
  CF_Write_Sector(Sector, Buffer);
  Result := true;
{$ENDIF}

{$IFDEF IDE}
  Result := IDE_WriteSector(Sector, Buffer);
{$ENDIF}
end;
```

Depending on the "DEFINED" version, the routines will work for MMC, SDMMC (drivers from Yo2Lio), CF or IDE.

The signature of the **Medium_Size** function is:

`function Fat32_Dev_Capacity_Sectors: DWord;`
Returns the size, in sectors, of the medium (e.g. SD/mmc card).

Example (only SD/MMC cards for the moment):

```
{$DEFINE MMC}

function Fat32_Dev_Capacity_Sectors: DWord; // medium size in sectors (used by "Fat32_Format")
begin
  Result := 0;
{$IFDEF MMC}
  Result := SDMMC_CardSize_Sectors; // uses the units "SDMMC_Utils_mmc" and "BitUtils"
{$ENDIF}
end;
```

8 Fat32 library Initialisation

The initialisation consists of 4 parts:

- The initialisation of the **storage medium**
- The definition of the **File Variable**(s) (only for **Fat32_2** and **Fat32_2_MD**)
- The initialisation of the **Fat32 library**
- The initialisation of the **File Variables** (only for **Fat32_2** and **Fat32_2_MD**)

8.1 Initialisation of the storage medium

Important:

"**xxx_init**" (xxx_init being the init routine for the hardware, e.g. "mmc_init") must be **called** (with success) **before** "Fat32_Init" can be called. The user of the Fat32_1 or Fat32_2 library is responsible for the initialisation of the media containing the Fat32 system and file.

8.2 Initialisation of the Fat32 library

The library itself is initialised with the “Fat32_Init” function:

`function Fat32_Init: boolean;`
Initialises the Fat32 file system, reads in the basic card/disk data (Fat boot record), returns true if success, false on failure. IMPORTANT: "mmc_Init" (or equivalent) must be done (with success) before this routine can be called!

8.3 Initialisation of the File variables

The File variables are initialised with this procedure:

`procedure Fat32_File_Init(var FileVar: TFileVar);`
Initialises the FileVar record, the directory in it is set the main directory of the Fat32 device. Important: to be called right after "Fat32_Init" for each variable of type TFileVar.

8.4 Example

Example of the whole initialisation for an SD/MMC card as storage:

```
// definition of the “File variables”
var File1, File2, File3, File4: TFileVar;  // e.g. 4 files can be opened simultaneously
    Success: boolean;
...

  // ------------------ low speed SPI initialisation ---------------------------------

  SPI1_Init_Advanced(_SPI_MASTER_OSC_DIV64, _SPI_DATA_SAMPLE_MIDDLE, _SPI_CLK_IDLE_LOW,
_SPI_LOW_2_HIGH);
  delay_ms(250);

  // -------------------- SD/MMC card initialisation ---------------------------------

  Success := (0 = Mmc_Init);

  if Success then
  begin
    Uart_Write_Line('mmc_init success');
  end
  else
  begin
    Uart_Write_Line('mmc_init failed');
    while true do;   // halt here if the mmc card could not be initialised
  end;

  // ------------------ high speed SPI initialisation ---------------------------------

  SPI1_Init_Advanced(_SPI_MASTER_OSC_DIV4, _SPI_DATA_SAMPLE_MIDDLE, _SPI_CLK_IDLE_LOW,
_SPI_LOW_2_HIGH);


  // ---------------- The Fat32 library initialisation ------------------------------

  if Fat32_Init then  // here "File1" is used for fat init purposes
  begin
    Uart_Write_Line('Fat32_init success');
  end
  else
  begin
```

```
    Uart_Write_Line('Fat32_init failed');
    while true do; // halt here if the no Fat32 is detected
  end;

// -------------- File variable initialisation (only for Fat32_2) ---------------------
  Fat32_File_Init(File1);
  Fat32_File_Init(File2);
  Fat32_File_Init(File3);
  Fat32_File_Init(File4);

// --------------------------- Library ready for usage --------------------------
```

Next to the library also the *File variables* must be initialised with the “Fat32_File_Init” routine (see above example).

9 File related functions

All file operations take place in the *current directory* of the *File variable* associated with a file. Only one file can be opened for every *File variable*.
After a files has been *opened*, *read* from and/or *written* to it has to be *closed* also, otherwise data could be lost.

9.1 The File Pointer

Files are “random access”, the place in the file to be read from or written to can be chosen by manipulating the *FilePointer*. Of course a file can also be accessed sequentially by not manipulating the *FilePointer* yourself.

The *FilePointer* is an internal Fat32 variable pointing to the *next place (byte) in the file that will be written to or read from*. Most file operations use and update this *FilePointer*.
The range of the *FilePointer* is from zero (beginning of the file) to the size of the file (in bytes) minus 1.
The only time the *FilePointer* can have the value “size of the file (in bytes)” is when appending data to the end of the file.

The *FilePointer* of open files (see below how to open a file) can be handled by the user with:

Set the file pointer to a certain position:

`procedure Fat32_Seek(var FileVar: TFileVar; Position: DWord);`
Sets the file pointer of the currently open file to "Position". If "Position" is outside the file, then it becomes the same as with "Append"

Reset the file pointer to zero (beginning of the file):

`procedure Fat32_Reset(var FileVar: TFileVar; var _Size: DWord);`
Resets the file pointer of the currently assigned file to zero (first byte of the file).Upon exit, "_Size" holds the filesize in bytes.

As you can see, the latter procedure also gives back the size of the file (in bytes) in the “_Size” parameter.

Go to the end of the existing file to append new data to it. This is done with:

`procedure Fat32_Append(var FileVar: TFileVar);`
Sets the file pointer of the currently assigned file to the next place after its last byte.

Go to a sector boundary (can be used with Fat32_Read_Sector or Fat32_Write_Sector):

`procedure Fat32_Seek_Sector(var FileVar: TFileVar; Sector: DWord);`
Sets the file pointer of the currently open file to "Sector * BytesPerSector". If "Sector * BytesPerSector" is outside the file, then it becomes the same as with "Append"

Go to the end of the existing file to append a new sector to it:

`procedure Fat32_Append_Sector(var FileVar: TFileVar);`
Sets the file pointer of the currently assigned file to the next place after its last sector

Get the current value of the file pointer:

`function Fat32_FilePointer(var FileVar: TFileVar): DWord;`
Returns the file pointer value of the currently open file. The file pointer is the byte number in the file that will be read from or written to next.

For examples, see section Random file access.

9.2 Opening a file

A file is always opened with the function:

`function Fat32_Assign(var FileVar: TFileVar; var LongFn: TLongFileName; file_cre_attr: byte): boolean;`
Opens a file with name "LongFn" and returns true on success. Only files can be opened, no directories, VolumeId's etc... The file is created (if not already existing) provided "file_cre_attr" contains "faCreate". The File pointer (points to the next byte to be read or written) is set to zero.

As you can see the *File variable*, the (wanted) filename and an “attribute” have to be provided. If you want the file to be created if it does not exist, then provide “faCreate” as attribute, else provide zero.

A file should always be opened (or “assigned”) before it can be read or written to. Also for some other actions the file has to be open.

Examples:

```
If Fat32_Assign(File1, 'TestFile.txt', 0) then
begin
   // the file is opened successfully
end else
begin
   // The file could not be opened (perhaps it does not exist)
end;
```

```
If Fat32_Assign(File1, 'TestFile.txt', faCreate) then
begin
   // the file is opened successfully, it did not exist it has been created
end else
begin
   // The file could not be opened or not created
end;
```

9.3 Closing a file

Closing a file is very straightforward:

`procedure Fat32_Close(var FileVar: TFileVar);`
Closes the currently assigned file (flushes the data buffer etc...) Important: Always to be called when finishing using a file, except when using a swap file or "Fat32_Flush" was called after the last write action.

Closing (or Flushing) a file that is not needed any more is obligatory.

Example:

```
If Fat32_Assign(File1, 'TestFile.txt', faCreate) then
begin
   // the file is opened successfully, it did not exist it has been created
   // do handle the file content
   Fat32_Close(File1);
end else…
```

See also the routine Fat32_Flush about another manner of "closing" a file.

9.4 Rewriting a file

After opening a file, one can choose to use the already existing content of the file, or clear the file and start with an empty one. The latter is done with:

`procedure Fat32_Rewrite(var FileVar: TFileVar);`
Discards the content of the currently assigned file (as if it was newly created), and sets its filesize and FilePointer to 0.

Example:

```
If Fat32_Assign(File1, 'TestFile.txt', faCreate) then
begin
  Fat32_Rewrite(File1); // the file will be emptied here before further processing.
  …
  Fat32_Close(File1);
end;
```

9.5 Reading from a file

Most file reading procedures use and update the *FilePointer*. Before reading you can manipulate this pointer with the procedures in section The FilePointer.
Reading from an open file can be done with a number of routines:

9.5.1 Reading one byte

Reading one byte from a file:

`procedure Fat32_Read(var FileVar: TFileVar; var _Data: byte);`
Reads 1 byte out of the currently assigned file into "_Data". On exit, the CurrentFilePointer points to the next byte in the file to be read.

Example:

```
Var Ch: Char;
…
If Fat32_Assign(File1, 'TestFile.txt', faCreate) then
Begin
  …
  Ch := Fat32_Read(File1); // read one byte/character from the file
  …
  Fat32_Close(File1);
end;
```

9.5.2 Reading more bytes

More bytes (a "buffer") can be read by using the following function:

`function Fat32_ReadBuffer(var FileVar: TFileVar; var Buffer: array[4096] of byte; DataLen: Word): word;`
Reads at most "DataLen" bytes out of the currently assigned file into "Buffer". Upon exit, the CurrentFilePointer points to the next byte in the file to be read. Returns the actual number of bytes read (reading beyond EOF is not done).

Example:

```
Var Buffer: array[100] of byte;
    Nr: Dword;
…
If Fat32_Assign(File1, 'TestFile.txt', faCreate) then
Begin
  …
  Nr := Fat32_ReadBuffer(File1, Buffer, SizeOf(Buffer));
  // try to read 100 bytes/characters from the file
  // After the call "Nr" will hold the number of bytes actually read
  //(reading beyond end of file is not done)
  …
  Fat32_Close(File1);
end;
```

The following procedure is a "Sector" based procedure, not used in every day activities but added for completeness sake.

9.5.3 Reading a complete file sector

A complete sector of a file (512 bytes) can be read by using:

`function Fat32_Read_Sector(var FileVar: TFileVar; var Buffer: array[512] of byte): DWord;`
Reads one sector (if possible) out of the currently open file at position "CurrentFilePointer" to "Buffer". Returns the actual number of bytes read (0..512). Afterwards "CurrentFilePointer" points to the next byte in the file to be read. Attention: "CurrentFilePointer" mod 512 will be set to zero (so, at a sector boundary) when "Fat32_Read_Sector" is called!!!!

Example:

```
Var Buffer: array[512] of byte;
```

```
…
If Fat32_Assign(File1, 'TestFile.txt', faCreate) then
Begin
  …
  While not Fat32_EOF(File1) do
  begin
    Fat32_Read_Sector(File1, Buffer); // read all sectors of the file one after another
    // process each sector here
  end;
  …
  Fat32_Close(File1);
end;
```

Random access reading is possible by calling Fat32_Seek_Sector before **Fat32_Read_Sector.**

The functionality of **Fat32_Read_Sector** is the same as Fat32_ReadBuffer, but more efficient due to the sector alignment.

9.5.4 End Of File indication

There is one special function, usually used when "reading" a file's content, indicating that the **E**nd **o**f **F**ile has been reached:

`function Fat32_EOF(var FileVar: TFileVar): boolean;`
Returns true on an end-of-file condition: the file pointer is outside the file. During "appending" data EOF is always true.

Example:

```
Var Ch: Char;
…
If Fat32_Assign(File1, 'TestFile.txt', faCreate) then
Begin
  …
  while not(Fat32_EOF(File1)) do
  begin
    Ch := Fat32_Read(File1); // read one byte/character from the file until all done
    // process here the byte/character read from the file
  end;
  …
  Fat32_Close(File1);
end;
```

9.6 Writing to a file

All file writing procedures use and update the *FilePointer*. Before writing you can manipulate this pointer with the procedures in section The FilePointer.

Also using the Fat32_Append procedure (see below) manipulates the *FilePointer*:

9.6.1 Appending to a file

After opening a file, one can choose to go to the end of the existing file to append new data to it. This is done with:

`procedure Fat32_Append(var FileVar: TFileVar);`
Sets the file pointer of the currently assigned file to the next place after its last byte.

Example:

```
Var Ch: Char;
…
If Fat32_Assign(File1, 'TestFile.txt', faCreate) then
Begin
  Fat32_Append(File1); // the next write (and read) action will occur at the end
                       // of the file
  Fat32_Write(File1, 'A'); // the character 'A' will be appended to the file
  // process here the file further
  Fat32_Close(File1);
end;
```

Fat32_Append is the same as "Fat32_Seek(File1, Fat32_Get_File_Size(File1));

Setting the file pointer to append a whole sector to a file is done with:

`procedure Fat32_Append_Sector(var FileVar: TFileVar);`
Sets the file pointer of the currently assigned file to the next place after its last sector.

Fat32_Append_Sector is the same as
"Fat32_Seek_Sector(File1, Fat32_Get_File_Size_Sectors(File1))";

9.6.2 Writing one byte

`procedure Fat32_Write(var FileVar: TFileVar; _Data: byte);`
Writes 1 byte ("_Data") to the currently assigned file. Upon exit, the CurrentFilePointer points to the next byte in the file to be written.

Example:

```
Var Ch: Char;
…
If Fat32_Assign(File1, 'TestFile.txt', faCreate) then
Begin
  …
  Fat32_Write(File1, 'A'); // the character 'A' will be written to the file
                           // at the "FilePointer" position. The latter will be
                           // incremented by one.
  // process here the file further
  Fat32_Close(File1);
end;
```

9.6.3 Writing more bytes

`procedure Fat32_WriteBuffer(var FileVar: TFileVar; var Buffer: array[4096] of byte; DataLen: Word);`
Writes "DataLen" bytes out of "Buffer" to the currently open file at position "CurrentFilePointer". Afterwards CurrentFilePointer points to the next byte in the file to be written.

Example:

```
Var Buffer: array[100] of byte;
…
If Fat32_Assign(File1, 'TestFile.txt', faCreate) then
Begin
  …
  // fill the buffer with data to be written
  Fat32_WriteBuffer(File1, Buffer, 50);// the first 50 bytes of the buffer will be written
                                       // to the file at the "FilePointer" position.
                                       // The letter will be incremented by 50.
  …
```

```
  Fat32_Close(File1);
end;
```

9.6.4 Writing a string

procedure Fat32_WriteText(var FileVar: TFileVar; var S: string[4095]);
Writes string "S" to the currently open file at position "CurrentFilePointer", no CR LF is written after the string. Afterwards CurrentFilePointer points to the next byte in the file to be written.

Writes a string to the open file without <CR><LF>

Example:

```
Var Str1: string[10];
…
If Fat32_Assign(File1, 'TestFile.txt', faCreate) then
Begin
  …
  Str1 := 'Text';
  Fat32_WriteText(File1, Str1); // the characters 'Text' will be written to the file
                                // at the "FilePointer". The latter will be incremented
                                // by 4.
  Fat32_WriteText(File1, 'abcde'); // the characters 'abcde' will be written to the file
                                   // at the "FilePointer". The latter will be incremented
                                   // by 5.
  …
  Fat32_Close(File1);
end;
```

Writing a string followed by CRLF to a file is done with:

procedure Fat32_WriteLine(var FileVar: TFileVar; var S: string[4095]);
Writes string "S" to the currently open file at position "CurrentFilePointer", CR LF additionally written after the string. Afterwards CurrentFilePointer points to the next byte in the file to be written.

Writes a string to the file followed by <CR><LF>

Example:

```
Var Str1: string[10];
…
If Fat32_Assign(File1, 'TestFile.txt', faCreate) then
Begin
  …
  Str1 := 'Text';
  Fat32_WriteLine(File1, Str1); // the characters 'Text<cr><lf>' will be written to the
                                // file at the "FilePointer" position. The latter will be
                                // incremented by 6.
  Fat32_WriteLine(File1, 'abcde');// the characters 'abcde<cr><lf>' will be written to the
                                  // file at the "FilePointer" position.
                                  // The latter will be incremented by 7.
  …
  Fat32_Close(File1);
end;
```

9.6.5 Writing constant data

Following procedure writes data defined as “const” to a file:

`procedure Fat32_Write_Const_Buffer(var FileVar: TFileVar; const _Data: ^byte; Len: word);`
Writes "Len" bytes out of "_Data" (constant data) to the currently open file at position "CurrentFilePointer". Afterwards CurrentFilePointer points to the next byte in the file to be written. Usage: Fat32_Write_Const_Buffer(@Constant, NrofConstantbytes);

Example:

```
const Str1: string[23] = 'Size    Attrs  FileName';
…
If Fat32_Assign(File1, 'TestFile.txt', faCreate) then
Begin
  …
  Fat32_Write_Const_Buffer(File1, @Str1, 23); // 23 characters from Str1 will be
                                              // written to the file at the "Filepointer"
                                              // position. The latter will be incremented
                                              // by 23.
  …
  Fat32_Close(File1);
end;
```

The 2 following procedures are “Sector” based procedures, not used in every day activities but added for completeness sake.

9.6.6 Writing a complete file sector

A complete sector (512 bytes) can be written/added to a file by using:

`procedure Fat32_Write_Sector(var FileVar: TFileVar; var Buffer: array[512] of byte);`
Writes 512 bytes out of "Buffer" to the currently open file at position "FilePointer". Afterwards "FilePointer" points to the next byte in the file to be written. Attention: "FilePointer" mod 512 must be zero (so, at a sector boundary) when "Fat32_Write_Sector" is called!!!!

This procedure can be used to write to or extend the file with one sector. It uses and updates the *FilePointer*. Important: the *FilePointer* must always be at a sector boundary before the call to the routine!

Example:

```
Var Nr: Dword;
    Buffer: array[512] of byte;
…
If Fat32_Assign(File1, 'TestFile.txt', faCreate) then
Begin
  // here the FilePointer is zero (sector boundary)
  for Nr := 0 to 9 do // will write 10 sectors to the file
  begin
    // fill the fuffer with data to be written to the file here
    Fat32_Write_Sector(File1, Buffer); // write one sector to the file one at the
                                       // "FilePointer" position. The latter will
                                       // be incremented with 512.
  end;
  …
  Fat32_Close(File1);
end;
```

Random access writing is possible by calling Fat32_Seek_Sector before **Fat32_Write_Sector.**

The functionality of Fat32_Write_Sector is the same as Fat32_WriteBuffer, but more efficient due to the sector alignment.

9.7 Sequential file access

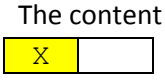
Sequential file access is done by simply using the Fat32_Assign, Fat32_Append, Fat32_Reset, Fat32_Rewrite, Fat32_Write, Fat32_WriteBuffer, Fat32_Write_Const_Buffer, Fat32_WriteText or Fat32_WriteLine as defined above, without manipulation of the *Filepointer*.

All data is written to or read from the "next" position in the file. With Fat32_Reset one can start from the beginning of the file again to read or write.
One can not read beyond the last byte in the file (end of file reached), but writing is possible: to append data to the end of the file.

Example (Fat32 related calls are in blue):

```
Var Buffer1, Buffer2, Buffer3: array[10] of byte;
    Str1: string[5];
    Ch: char;
    FileSize: Dword;
…
Memset(@Buffer1, 'A', SizeOf(Buffer1)); // fill the buffer with all "A"'s
Memset(@Buffer2, 'B', SizeOf(Buffer2)); // fill the buffer with all "B"'s
Str1 := 'abcde';
…
Fat32_Assign(File1, 'TestFile.txt', faCreate); // the file will be opened and created if
necessary

Fat32_Rewrite(File1);     // the file will be emptied

Fat32_Write(File1,'X');  // will write an 'X' in position zero of the file, the next position
to write to (or read from)(the Filepointer)is position 1. The file is 1 byte long now.
```

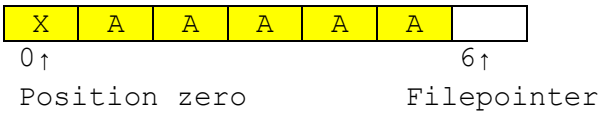
The content of the file looks now:



```
0↑  1↑
 ↑   └ Filepointer
Position zero
```

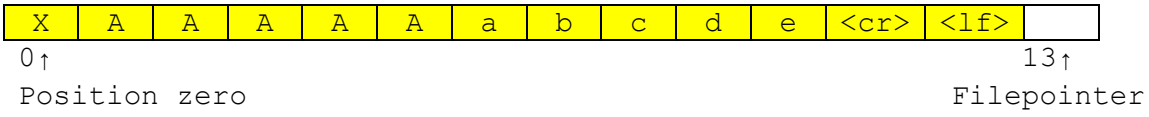
```
Fat32_WriteBuffer(File1, Buffer1, 5); // will write 5 "A"'s to the file, the next position in
the file to write to (or read from) (the file pointer) is 6. The file is 6 bytes long now.
```

The content of the file looks now:



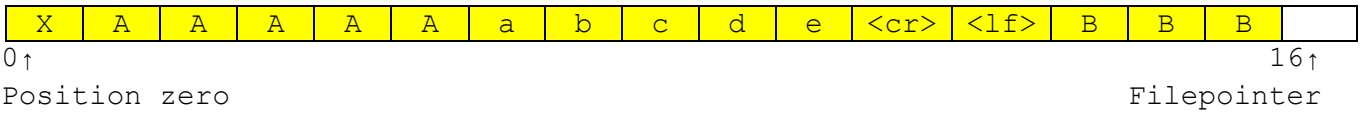

```
Fat32_WriteLine(File1, Str1); // writes 'abcde' followed by CR LF to the file. The Filepointer
is now at position 13. The file is 13 bytes long.
```

The content of the file looks now:




```
Fat32_WriteBuffer(File1, Buffer2, 3); // will add 3 "B"'s to the file. The Filepointer is now
at position 16. The file is 16 bytes long now.
```

The content of the file looks now:

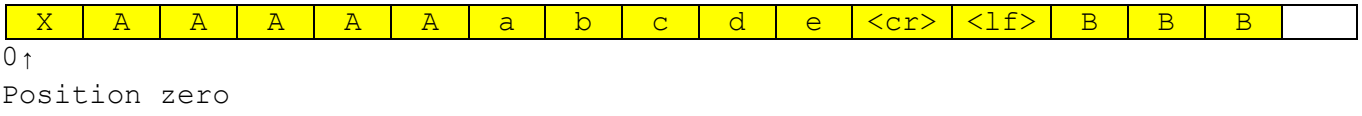


and the Filepointer points after the last "B" (position 16). (remark: <cr> and <lf> are each only 1 character or byte).

Let's now read some data:

```
Memset(@Buffer3, 0, sizeOf(Buffer3)); // clear buffer 3

Fat32_Reset(File1 , FileSize);  // we want to start reading from the beginning of the file
```

The content of the file looks now:




```
Fat32_ReadBuffer(File1, Buffer3, 4); // read 4 bytes from the file. Those 4 bytes are
Read from the file into buffer3. The context of buffer3 will be: XAAA… and the file pointer
will be on position 4.
```

The content of the file looks now:

X	A	A	A	A	A	a	b	c	d	e	<cr>	<lf>	B	B	B	

0↑ 4↑
Position zero Filepointer

```
Fat32_Read(File1, Ch); // will read 1 character from the file into variable "Ch". Its content
will become 'A'. The file pointer will now point to 5.
```

The content of the file looks now:

X	A	A	A	A	A	a	b	c	d	e	<cr>	<lf>	B	B	B	

0↑ 5↑
Position zero Filepointer

```
Fat32_ReadBuffer(File1, Buffer3, SizeOf(Buffer3)); // will read 10 bytes from the file,
starting at position 5 into Buffer3. The content of Buffer3 will become:
Aabcde<cr><lf>BB and the file pointer will point to the last "B" in the file (position 15).
```

The content of the file looks now:

X	A	A	A	A	A	a	b	c	d	e	<cr>	<lf>	B	B	B	

0↑ 15↑
Position zero Filepointer

```
Fat32_Reset(File1, FileSize); // we will start over, the file pointer is zero again
Fat32_ReadBuffer(File1, Buffer3, 5); // the file pointer now points to position 5.
```

The content of the file looks now:

X	A	A	A	A	A	a	b	c	d	e	<cr>	<lf>	B	B	B	

0↑ 5↑
Position zero Filepointer

```
Fat32_WriteText(File1, 'Test'); // the text 'Test' is written from position 5 onwards and
replaces the original test.
```

The content of the file looks now:

X	A	A	A	A	T	e	s	t	d	e	<cr>	<lf>	B	B	B	

0↑ 9↑
Position zero Filepointer

```
Fat32_Close(File1); // finally close the file.
```

In all above "file contents" the "File" content is marked with yellow.

An actual application example, the routine below dumps the content of an (ascii) file to the uart:

```
procedure DumpFile(var MyFile: TFileVar; var Name: TLongFileName);
var Ch: char;
    Bytes, I: DWord;
begin
  if Fat32_Assign(MyFile, Name, 0) then
  begin
    Fat32_Reset(MyFile, Bytes);
    if Bytes > 0 then
```

```
    begin
      for I := 0 to Bytes - 1 do
      begin
        Fat32_Read(MyFile, Ch);
        Uart1_Write(Ch);
      end;
    end;
  end;
  Uart_Write_Line(' ');
end;
```

9.8 Random file access

As explained already, random access let's the using program decide where to read and/or write in the file. As long as the action occurs within the file boundaries that decision is respected, the content is read (read action) or replaced (write action). When trying to read outsize the file (the Filepointer is beyond the last byte of the file) then nothing will be read.
If the using program tries to write beyond end of file the Filepointer is changed (if needed) to point to the next byte with respect to the last byte of the file (see below for example).

Example (Fat32 related calls are in blue):

```
var File1: TFileVar; // The "File Variable(s)"

Var Buffer1: array[100] of byte;
    I: word;
    Success: boolean;
    FileSize, Nr: DWord;
…
  for I := 0 to 9 do Buffer1[I] := I + 48;
  // fill Buffer1 with character I ('0123456789');

  Fat32_Assign(File1, 'Random.txt', faCreate); // Create the file if necessary
  Fat32_Rewrite(File1); // empty the file

  for I := 1 to 3 do
  Fat32_WriteBuffer(File1, Buffer1, 10); // sequential write of 30 characters.

  Fat32_Close(File1);

  Uart_write_line('File Content:');      // The file contents is now:
                                         // "012345678901234567890123456789"
  DumpFile(File1, 'Random.txt');         // see DumpFile
  Uart_write_line('');

  Memset(@Buffer1, 0, SizeOf(Buffer1));
  Fat32_Assign(File1, 'Random.txt', 0);
```

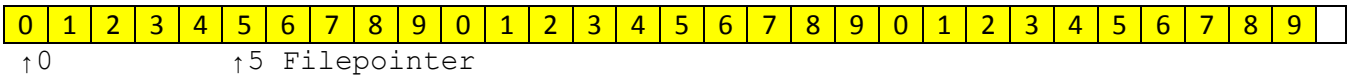
The file looks like this now:

0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	

```
↑0
|Position zero
Filepointer
```

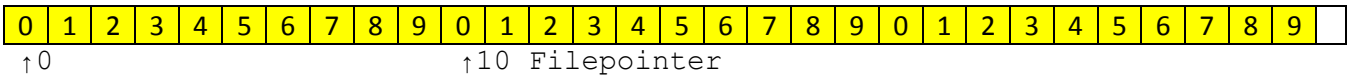
```
  Fat32_Seek(File1, 5);                        // goto position 5 in the file
```

The file looks like this now:



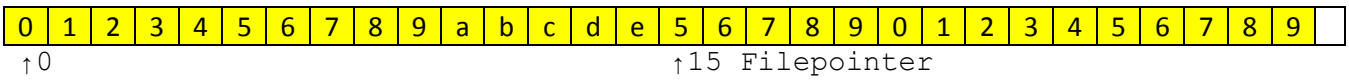

```
  Fat32_ReadBuffer(File1, Buffer1, 10);        // read in 10 bytes in the buffer

  DumpBuffer(Buffer1, 10, 'Buffer1 after reading from position 5: ');
  // The buffer's contents is "5678901234"

  Fat32_Seek(File1, 10);                       // points to the second "0"
```

The file looks like this now:



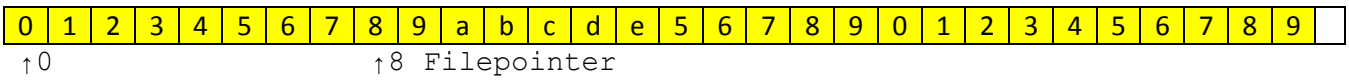

```
  Fat32_WriteText(File1, 'abcde');             // replaces contents with "abcde"
```

The file looks like this now:



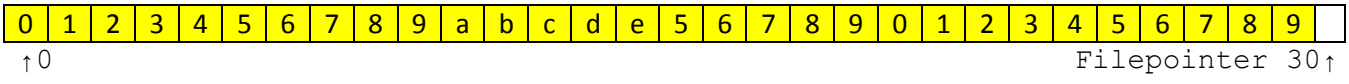

```
  Memset(@Buffer1, 0, SizeOf(Buffer1));
  Fat32_Seek(File1, 8);                        // Points to the first "8" now
```

The file looks like this now:



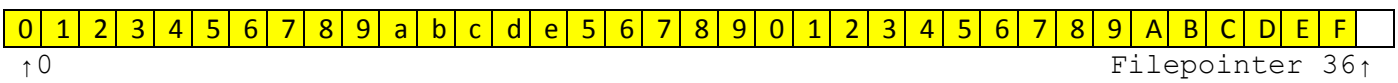

```
  Fat32_ReadBuffer(File1, Buffer1, 15);        // The content of the buffer will be
                                               // "89abcde56789012"
  DumpBuffer(Buffer1, 15, 'Buffer1 after writing: ');

  Fat32_Append(File1);                         // This will make the file pointer point
                                               // one byte past the end-of-file
```

The file looks like this now:




```
  Fat32_Writetext(File1, 'ABCDEF');
```

The file looks like this now:



```
  Fat32_Seek(File1, 25);                        // will make the file pointer point to
                                                // the last "5" in the file
```

The file looks like this now:

↑0 Filepointer 25↑
Position zero

```
  Nr := Fat32_ReadBuffer(File1, Buffer1, SizeOf(Buffer1));
  // will read up to 100 bytes from position 25 onwards
```

The file looks like this now:

↑0 Filepointer 36↑
Position zero

```
  Uart_Write_Text('Nr of bytes read from position 25 onwards: ');
  Uart_Write_Line_Dword(Nr);
  DumpBuffer(Buffer1, Nr, 'Buffer1 after appending: ');
  // The buffer content will be: "56789ABCDEF"  (only 11 bytes read)

  Fat32_Close(File1);

  Uart_write_line('');
  Uart_write_line('File Content:');
  DumpFile(File1, 'Random.txt');            // see DumpFile
                                            // The file contents is now:
                                            //"0123456789abcde567890123456789ABCDEF"

  Fat32_Assign(File1, 'Random.txt', 0);
```

The file looks like this now:

↑0
|Position zero
Filepointer

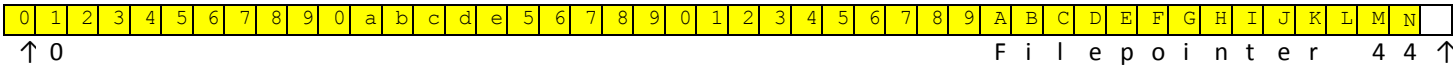
```
  Fat32_Seek(File1, 100);                   // seek (far) outside the file
```

The file looks like this now:

↑0 Filepointer 36↑
Position zero

```
  Uart_write_text('Filepointer after seek (100): ');
  // The file pointer should point to end-of-file plus one (as for append)
  Uart_write_line_Dword(Fat32_FilePointer(File1));

  Fat32_writeText(File1, 'GHIJKLMN');
```

The file looks like this now:



↑ 0 F i l e p o i n t e r 4 4 ↑

```
  Fat32_Close(File1);

  Uart_write_line('');
  Uart_write_line('File Content:');
  DumpFile(File1, 'Random.txt'); // see DumpFile
  // The file contents is now: "0123456789abcde567890123456789ABCDEFGHIJKLMN"
```

9.9 Multiple File access

This is only possible with the Fat32_2 version.

Here is an example of how to access more than one file at a time. The example creates 2 files and fills them with some data. After that a third file is created and filled with the data (byte per byte) alternating from file 1 and file 2. All files are in different directories.

```
var File1, File2, file3: TFileVar; // the "file variables"
...
  // make 2 files and fill them with data
  Fat32_MkDir_ChDir(File1, 'Directory_One');
  if Fat32_Assign(File1, 'TestFile1.txt', faCreate) then
  begin
    Uart_write_Line('Filling file 1');
    For I := 'A' to 'z' do Fat32_Write(File1, I);
    Fat32_Close(File1);
  end;

  Fat32_MkDir_ChDir(File2, 'Directory_Two');
  if Fat32_Assign(File2, 'TestFile2.txt', faCreate) then
  begin
    Uart_write_Line('Filling file 2');
    For J := 0 to 5 do For I := '0' to '9' do Fat32_Write(File2, I);
    Fat32_Close(File2);
  end;

  // create the third file and fill it with data from the first two
  // the data is taken alternating from file 1 and file 2

  Fat32_MkDir_ChDir(File3, 'Directory_Three');
  if Fat32_Assign(File3, 'Destination.txt', faCreate) then
  begin
    Fat32_Assign(File1, 'TestFile1.txt', 0); // open file 1
    Fat32_Assign(File2, 'TestFile2.txt', 0); // open file 2

    while (not Fat32_Eof(File1)) or (not Fat32_Eof(File2)) do
    // as long as one of the files is not done completely
    begin
      if not Fat32_Eof(File1) then
      begin
        Fat32_Read(File1, Ch);
        Fat32_Write(File3, Ch);
      end;

      if not Fat32_Eof(File2) then
      begin
        Fat32_Read(File2, Ch);
        Fat32_Write(File3, Ch);
      end;
    end;

    Fat32_Close(File1);
    Fat32_Close(File2);
    Fat32_Close(File3);
```

```
  end;
```

After execution of the above code, the contents of the files is:

TestFile1.txt: `ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz`

TestFile2.txt: `01234567890123456789012345678901234567890123456789012345678901234567890123456789`

Destination.txt:

`A0B1C2D3E4F5G6H7I8J9K0L1M2N3O4P5Q6R7S8T9U0V1W2X3Y4Z5[6\7]8^9_0`1a2b3c4d5e6f7g8h9i0j1k2l3m4n5o6p7q8r9s0t1u2v3w4x5y6z789`

9.10 Deleting File(s)

9.10.1 One File

`function Fat32_Delete(var FileVar: TFileVar; var Name: TLongFileName): boolean;`
Deletes file with "Name". Returns true if Success, else false. Attention! !!! this function closes first the currently open file !!!

This is the same function as the one for deleting directories.

Example:

```
Var Success: boolean;
…
If Fat32_Assign(File1, 'TestFile.txt', faCreate) then
Begin
  // write to the file, read from it…
  Fat32_Close(File1);
end;

Success := Fat32_Delete(File1, 'TestFile.txt');
if Success then …
```

9.10.2 All Files in a directory

`procedure Fat32_Delete_Files(var FileVar: TFileVar);`
Deletes all "Files" in the current directory. The sub directories in the current directory are not deleted.

Example:

```
Var Success: boolean;
…
Success := Fat32_Chdir(File1, 'Directory_One');
if Success do
begin
   Fat32_Delete_Files(File1); // all files in 'Directory_One' are deleted, its
                              // subdirectories are unaltered
   // …
end;
```

9.10.3 All Files and all subdirectories (recursively) in a directory

Deleting all files and all subdirectories recursively (meaning also all files and subdirectories of the subdirectories etc. are deleted) is done with:

`function Fat32_Delete_All(var FileVar: TFileVar): boolean;`
Empties the current directory: all files and subdirs are removed. Returns true if success (the all flies and directories were removed or it did not exist already). Attention! !!! this function closes first the currently open file!!!

Example:

```
Var Success: boolean;
…
Success := Fat32_Chdir(File1, 'Directory_One');
if Success do
begin
   Fat32_Delete_All(File1); // all files and subdirectories in 'Directory_One' are
                            // deleted, the directory is empty
   // …
end;
```

9.11 Renaming a file

Renaming a file is very simple, done with:

`function Fat32_Rename(var FileVar: TFileVar; var OldName, NewName: TLongFileName): boolean;`
Renames the file named "OldName" to "NewName" in the current directory. Returns "true" if successful, else "false" (e.g. "OldName" does not exists or "NewName" already exists). Attention! !!! this function closes first the currently open file !!!

Example:

```
Var Success: boolean;
…
If Fat32_Assign(File1, 'TestFile.txt', faCreate) then
Begin
  // write to the file, read from it…
  Fat32_Close(File1);
end;

Success := Fat32_Rename(File1, 'TestFile.txt', 'AnOtherFileName.txt);
if Success then …
```

9.12 Finding files or directories

All following routines return “true” if a file is found according the wanted criteria and “false” is no file exist (any more) according the wanted criteria.
If the function returns “true”, the data of the found file or directory are in the record “Fat32_DirItem” inside the *File variable* used.

The content of “Fat32_DirItem” is:

```
TFat32DirItem  =
record
  FileName      : TLongFileName;  // Long filename if there is one, else short FileName
  ShortFileName : TShortFileName; // short filename
  FileAttr      : byte;           // file attributes
  FileSize      : DWord;          // FileSize in bytes
  FindDirEntry  : DWord;          // for internal usage only
end;
```

Two kinds of “Find” routines exist:

- A kind that only takes the wanted “Attribute” as criterion
- A kind that takes both a filename and an attribute as criterion

For both kind the following holds: a new find action always starts with a call to Fat32_FindFirst_..., and all next calls have to be Fat32_FindNext_....

File attributes used the “Find” routines are:

```
faAnyFile   = $00;
faReadOnly  = $01; // bit 0
faHidden    = $02; // bit 1
faSysFile   = $04; // bit 2
faVolumeId  = $08; // bit 3
faDirectory = $10; // bit 4
faArchive   = $20; // bit 5
faFile[5]     = $40; // bit 6, not directory, not volumeId
```

`function Fat32_FindFirst(var FileVar: TFileVar; FileAttr: byte): boolean;`
Returns true if the routine finds the first file/directory (if any) and puts the result in "Fat32_DirItem". Only the current directory is searched. If no first file/directory present then the procedure returns false. To be called before "Fat32_FindNext" is used.

`function Fat32_FindNext(var FileVar: TFileVar; FileAttr: byte): boolean;`
Returns true if the routine finds a next file/directory (if any) and puts the result in "Fat32_DirItem". Only the current directory is searched. If no next file/directory present then the procedure returns false. Not to be called without a previous call to "Fat32_FindFirst".

[5] This attribute is a virtual one, it does not exist in the actual Fat32 file system.
It is only added here for ease of use.

Example: To find all directories in the current one:

```
var Found: boolean;
    DirName: TlongFileName;
…
Found := Fat32_FindFirst(File1, faDirectory);
while Found do
begin
  DirName := File1.Fat32_DirItem.FileName;
  Uart_Write_Line(DirName);  // send the found directory name to the uart for display
  Found := Fat32_FindNext(File1, faDirectory);
end;
```

`function Fat32_FindFirst_FN(var FileVar: TFileVar; var LongFN: TLongFileName; FAttr: byte): boolean;`
Same as "Fat32_FindFirst" but with filename ("LongFN") included in the search criteria. Allowed wildcard constructions in "LongFn": - "FileName.Ext" : Finds only the file/directory with the filename exactly equal to LongFn - "*.*" : Finds any file/directory (= same as "Fat32_FindFirst") - "File*.E*: : Finds all files/directories of which the filename starts with "File" and the extension starts with "E". Attention: The "*" can only be used to make the --> tail <-- of the filename or extension "don't care". "FileN?me.E?t : Finds all files/directories with the same name as LongFn, except the positions holding "?" which are don't care. "?" only represents 1 character! If "LongFn" has no dot ('.') in it, only files/directories with no extension are found.

`function Fat32_FindNext_FN(var FileVar: TFileVar; var LongFN: TLongFileName; FAttr: byte): boolean;`
Same as "Fat32_FindNext" but with filename ("LongFN") included in the search criteria. Allowed wildcard constructions in "LongFn": see "Fat32_FindFirst_FN".

Example: To find all “normal files” in the current directory with extension ‘.txt’:

```
var Found: boolean;
    FileName: TlongFileName;
…
Found := Fat32_FindFirst_FN(File1, '*.txt', faFile);
while Found do
begin
  FileName := File1.Fat32_DirItem.FileName;
  Uart_Write_Line(FileName);  // send the found file name to the uart for display
  Found := Fat32_FindNext_FN(File1, '*.txt', faFile);
end;
```

`function Fat32_FileExists(var FileVar: TFileVar; var LongFn: TLongFileName; FAttr: byte): boolean;`
Returns true if file with name "LongFn" and attribute "FAttr" exists. No backslashes allowed in "LongFn" (no multiple dirlevels).

Example to find out if 'Directory_One' resides in the current directory:

```
if Fat32_FileExists(File1, 'Directory_One', faDirectory) then
begin
  // do something
end;
```

9.13 Counting files

`function Fat32_FileCount(var FileVar: TFileVar; var Names: string[255]; Attr: byte): DWord;`
Count the number of files in the current directory of which the filename is in "Names" and the attributes comply with "Attr". "Names" is a comma separated list of ambiguous (wildcard) filenames, like '*.txt, *.log, File*.*', for the allowed wildcard constructions in the filenames: see "Fat32_FindFirst_FN". "FAttr" is any of the file types defined in unit "Fat32.mpas" Attention: * upon exit "Names" is an empty string! * the separate untrimmed filenames in "names" should not be longer than 30 characters.

With this routine a file/directory count can be done of files/directories of which the name is in a set of names. All files/directories counted must have a certain attribute.

Example to find all normal files of which the name starts with "Test" *or* have the extension ".txt":

```
var Str: string[50]
      Count: DWord;
...
Str := 'Test*,*.txt';
Count := Fat32_FileCount(File1, Str, faFile);
// do something with "Count"
```

9.14 Making Files with the directory content

Both following routines make a file in the current directory with the contents in it of the current directory (subdirs and files).
There is a version that generates a simple text file and a version that generated a html file. In the file also its own entry will be present, usually with a size of zero bytes.

`procedure Fat32_MakeDirFile(var FileVar: TFileVar; var DirFileName: TLongFileName);`
Makes a text file which holds the directory info (e.g. names and sizes of files present) of the current directory.

Example:

```
Fat32_MakeDirFile(File1, 'Dir.txt');
DumpFile(File1, 'Dir.txt'); // see DumpFile
```

The result might look like this:

```
Size     Attrs  FileName
----     -----  --------
0        D      .
0        D      ..
410      A      File_AA.txt
250      A      File_AB.txt
113      A      File_AC.txt
21       A      File_AD.txt
216      A      File_AE.txt
10       A      File_AF.txt
0        A      Dir.txt
```

`procedure Fat32_MakeDirFileHtm(var FileVar: TFileVar; var DirFileName: TLongFileName);`
Makes a html file which holds the directory info (e.g. names and sizes of files present) of the current directory.

```
Fat32_MakeDirFileHtm(File1, 'Dir.htm');
DumpFile(File1, 'Dir.htm'); // see DumpFile
```

The result might look like this:

```
<html><head><title>Files</title></head><body><pre>Size     Attrs  FileName
----     -----  --------
0        D      .
0        D      <A href="/DirectoryA/..">..</A>
410      A      <A href="/DirectoryA/File_longnameAA.txt">File_longnameAA.txt</A>
250      A      <A href="/DirectoryA/File_longnameAB.txt">File_longnameAB.txt</A>
113      A      <A href="/DirectoryA/File_longnameAC.txt">File_longnameAC.txt</A>
21       A      <A href="/DirectoryA/File_longnameAD.txt">File_longnameAD.txt</A>
216      A      <A href="/DirectoryA/File_longnameAE.txt">File_longnameAE.txt</A>
10       A      <A href="/DirectoryA/File_longnameAF.txt">File_longnameAF.txt</A>
327      A      <A href="/DirectoryA/Dir.txt">Dir.txt</A>
0        A      <A href="/DirectoryA/Dir.htm">Dir.htm</A>
</pre></body></html>
```

Which looks perfectly well when displayed by a browser and provides navigation to other directories and display (or execution) of the “files”.

9.15 Copying a file

`procedure Fat32_CopyFile(var SourceFile: TFileVar; var SourceFileName: string; var DestinationFile: TFileVar; var DestinationFileName: string);`
Copies file with name "SourceFileName" to a file with name "DestinationFileName". Attention! !!! this function closes first the currently open file(s) of "SourceFile" and "DestinationFile" !!!

Example:

```
var Source, Destination: TFileVar;
...
Fat32_File_Init(Source);
Fat32_File_Init(Destination);
...
Fat32_ChDir_FP(Source, '\Directory_One');      // goto the source directory
Fat32_ChDir_FP(Destination, '\Directory_Two'); // goto the destination directory
Fat32_CopyFile(Source, 'File_One.txt', Destination, 'Copy_of_File_One.txt');
...
```

A side effect of making a copy is that the copy is "defragmented". It is made using a swap file (see below).

9.16 Creating a SwapFile and use it

`function Fat32_Get_Swap_File(var FileVar: TFileVar; NoSectors: dword; var filename : TLongFileName; Attr : byte) : Dword;`
This function is used to create a Fat32 file of fixed size (NoSectors sectors) on the MMC/SD media, with consecutive sectors, making it possible to use direct sector read/write in the file without using the FAT32 file system any further. The function returns the number of the start sector for the newly created swap file, if there was enough free space on the MMC/SD/CF card or disk to create file of required size, 0 otherwise. Attention!!! If a file with specified name already exists on the media, it will be emptied, and a attempt will be made to re-use its space on the card/disk. No need to "close" the file after it was created with this function (the file is not open anyway from the file system's point of view). Afterwards the swap file can also be opened like a normal file with "Fat32_Assign", or its sectors can read from or written to directly.

A SwapFile is always created as a file of consecutive physical sectors. This means that one can read and write the file with the raw read/write sector commands of the medium (e.g. mmc card) involved if one knows the physical start sector of the file and the length of the file in sectors.

The function above makes a swap file of "NoSectors" and gives back the physical start sector of the file.

Example for SD/MMC cards:

```
var StartSector, Size,  Index: DWord;
      Buffer: array[512] of byte;
...
Size := 1000; // 1000 sectors wanted
StartSector := Fat32_Get_Swap_File(File1, Size, 'MySwapFile', 0);
if StartSector > 0 then // swap file of 1000 consecutive sectors could be made.
begin
  // writing to the file
  for Index := 0 to (Size - 1) do
  begin
    // fill the buffer here with data to be written
    mmc_write_sector(StartSector + Index, Buffer);
    ...
  end;
  // reading the file
    for Index := 0 to (Size - 1) do
  begin
    mmc_read_sector(StartSector + Index, Buffer);
    // handle here the buffer content read from the file
```

```
    ...
  end;
end;
```

As you can see the writing and reading to the file is done sector per sector in a random access manner. Since the native medium read/write routines are used here the speed is maximal.

Some remarks:

- The file should be created with a size that should be big enough for its entire lifespan. The size can not made bigger or smaller afterwards without losing the “swap file” criterion (consecutive sectors),
- You will have to remember the start sector and the number of sectors somewhere (e.g. eeprom) if you want to continue to use the file as a swap file,
- The file can however also be accessed by the normal Fat32 routines. It is then handled as a normal Fat32 file. All routines work as with any other file, BUT:
- *Extending the file with the normal Fat32 routines is also possible, but there is a chance that, after doing so, the file will be no “SwapFile” any more: the added sectors are most probably not consecutive any more.*

10 Directory related functions

The most common directory related functions are the following:

10.1 Making a directory

`function Fat32_MkDir(var FileVar: TFileVar; var LongFn: TLongFileName): boolean;`
Creates a directory inside the current one if it not already exists. No backslashes allowed in "LongFn" (no multiple dirlevels). Returns true if success (the directory was created or existed already). Attention! !!! this function closes first the currently open file !!!

Example:

```
for I := 'A' to 'D' do  // make some directories in the current directory
  begin

    DirName := 'Directory_';
    DirName[9] := I;        // directory names will be "DirectoryA", "DirectoryB", etc…

    uart_write_line('Making Dir '+ DirName);

    Success := FAT32_MkDir(File1, DirName);
    if (not Success) then
    begin
      uart_write_line('Could not make ' + DirName);
      while true do;
    end;
…
```

The example above makes a directory in the one pointed to by the *File variable* "File1" (further in the document called "*the current directory*" of a *File variable*.

10.2 Changing the directory

The current directory pointed to by a *File variable* (= the *current directory* of that *File variable*) can be changed with the routine:

`function Fat32_ChDir(var FileVar: TFileVar; var LongFn: TLongFileName): boolean;`
Changes directory to "LongFn" from within the current directory. No backslashes allowed in "LongFn" (no multiple dirlevels). Returns true if Success, else false. "Prevdir" can be used afterwards to return to the original directory (the one before "ChDir"). Attention! !!! this function closes first the currently open file !!!

Example (following the above one which made the directory):

```
  Success := FAT32_ChDir(File1, DirName); // goto the new directory
  if (not Success) then
  begin
    uart_write_line('Could not enter ' + DirName);
    while true do;
  end;
```

`Fat32_ChDir('\');` always changes the current directory back to the root.

`Fat32_ChDir('..');` always goes one directory level up.

There is also the possibility to go more than one level deeper or to change to an absolute path ("FP" stands for "Full Path"):

`function Fat32_ChDir_FP(var FileVar: TFileVar; var LongFn: TLongFileName): boolean;`
Changes directory to path "LongFn". Multiple dirlevels allowed, e.g. "\Directory1\Directory2", but: the different parts of the path themselves can not be longer than 128 bytes! Absolute paths start with "\", relative paths don't. '..' is allowed in the wanted directory. Returns true if Success, else false. "Prevdir" can be used afterwards to return to the original directory (the one before "ChDir_FP"). Attention! !!! this function closes first the currently open file !!!

Example:

```
Fat32_ChDir_FP(File1, '\DirLevel1\DirLevel2');  // start from the root and go 2 levels down.
```

A subsequent usage of Fat32_MkDir and Fat32_ChDir can be avoided with the usage of

`function Fat32_MkDir_ChDir(var FileVar: TFileVar; var LongFn: TLongFileName): boolean;`
Makes a directory and changes the current directory to it (same as subsequent "MkDir" and "ChDir").

Example:

```
Fat32_ChDir(File1, '\'); // go e.g. to the root directory
Fat32_MkDir_ChDir(File1, 'Directory_Three');  // creates and goes to 'Directory_Three'
```

Making a directory path (more than one directory level) and change to the last specified one can be made with:

`Function Fat32_MkDir_ChDir_FP(var FileVar: TFileVar; var LongFn: TLongFileName): boolean;`
Makes the full directory path specified in "LongFn", and changes the current directory to the last one.

Example:

```
Fat32_MkDir_ChDir_FP(File1, '\abc\def\ghi'); // create the directory path 'abc\def\ghi\'
                                             // in the root directory and change the current
                                             // directory to it.

Fat32_MkDir_ChDir_FP(File1,'abc\123'); // make the directory path 'abc\def' within the current
                                       // directory and change the current directory to it.
```

After usage of the Fat32_ChDir, Fat32_chDir_FP or Fat32_MkDir_ChDir routines, one can go back to the directory selected previous to their usage with:

`procedure Fat32_PrevDir(var FileVar: TFileVar);`
The current directory is changed back to the previously selected directory before the current one.

Example:

```
  // here we are in the "original directory"
  Fat32_ChDir_FP(File1, '\Directory1\Directory2\');
```

```
// do some work in this directory
Fat32_PrevDir(File1);  // do back to the original directory
```

There is also the possibility to “remember” a certain directory and select it later again with the routines

`procedure Fat32_PushDir(var FileVar: TFileVar);`
The current directory's start cluster is stored for "PopDir".

`procedure Fat32_PopDir(var FileVar: TFileVar);`
The current directory is changed back to the directory wherein the last "PushDir" was executed.

Example:

```
PushDir(File1); // remember the current directory
ChDir(File1, 'Dirname');
ChDir…
ChDir…
PopDir(File1); // go back to the "pushed" directory
```

10.3 Getting the current directory’s name

The name of the current directory can be fetched with

`procedure Fat32_Curdir(var FileVar: TFileVar; var CurrentDir: TLongFileName);`
Returns the name of the current directory in "CurrentDir". Attention! !!! This function closes first the currently open file !!! The actual variable used as CurrentDir must be (at least) of type string[255]!!!

which gives the name of the *current directory* of the *File variable*,
or

`procedure Fat32_Curdir_FP(var FileVar: TFileVar; var CurrentDir: TLongFileName);`
Returns the full path of the current directory in "CurrentDir". Attention! !!! This function closes first the currently open file !!! The actual variable used as CurrentDir must be (at least) of type string[255]!!!

which gives the *full path (FP)* of the *current directory* of the *File variable*.

The directory name is returned in the variable “CurrentDir”.

Example:

```
var FileName: TLongFileName;
...
Fat32_ChDir(File1, '\'); // go e.g. to the root
Fat32_MkDir_ChDir(File1, 'Directory_Three');
Fat32_MkDir_ChDir(File1, 'SubDirectory_One');
Fat32_CurDir(File1, FileName);  // the content of "FileName" will be "SubDirectory_One"
Fat32_CurDir_FP(File1, FileName);  // the content of "FileName" will be
"\Directory_Three\SubDirectory_One"
```

10.4 Renaming a directory

`function Fat32_Rename(var FileVar: TFileVar; var OldName, NewName: TLongFileName): boolean;`
Renames the file named "OldName" to "NewName" in the current directory. Returns "true" if successful, else "false" (e.g. "OldName" does not exists or "NewName" already exists). Attention! !!! this function closes first the currently open file !!!

Example:

```
Fat32_Rename(File1, 'Directory_One', 'Directory_Two');
```

Above example renames a subdirectory of the current directory ('Directory_One') into 'Directory_Two'.

10.5 Removing a directory

Is done with the procedure:

`function Fat32_RmDir(var FileVar: TFileVar; var LongFn: TLongFileName): boolean;`
Deletes a directory within the current one. No backslashes allowed in "LongFn" (no multiple dirlevels). Returns true if success (the directory was removed or it did not exist already). Attention!! Make sure the directory is empty (except for the '.' and '..' files), otherwise lost clusters will occur. This function closes first the currently open file !!!

Example:

```
Fat32_ChDir_FP(File1, '\Directory_One');
Fat32_RmDir(File1, 'SubDirectory_One');  // removes 'SubDirectory_One' in '\Directory_One'
```

Be careful: before using this procedure the directory to be removed must be empty (except for the '.' And '..' directories), otherwise "lost clusters" will occur.
This is the same routine as the one for removing files.

A routine that empties the directory to be removed (that is remove its subdirectories, their subdirectories etc…, and all files in those subdirs) prior to actually removing the directory itself, is:

`function Fat32_RmDir_All(var FileVar: TFileVar; var Fn: string): boolean;`
Deletes directory "Fn" - and all of its files, including subdirectories and all of their files - from the current directory. No backslashes allowed in "LongFn" (no multiple dirlevels). Returns true if success (the directory was removed or it did not exist already). Attention! !!! this function closes first the currently open file !!!

Using this routine no "lost clusters" will occur.

Example:

```
Fat32_ChDir_FP(File1, '\Directory_One');
Fat32_RmDir_All(File1, 'SubDirectory_One'); // removes recursively "SubDirectory_One"
```

11 Miscellaneous functions

11.1 Fat32 System routines

`function Fat32_Format(var VolumeLabel: string[11]): boolean;`
This routine creates a new boot sector (sector 0) and a new FSInfo sector. It deletes all files and directories and creates a new root directory. Returns true if Success, otherwise false. Also re-inits the Fat32 system (call to Fat32_Init). IMPORTANT: **do not forget to re-init the file variables afterwards**.

Example:

```
Fat32_Format('VolName'); // the volume label will be "VolName"
```

`function Fat32_QuickFormat(var VolumeLabel: string[11]): boolean;`
This routine deletes all files and directories and creates a new root directory. Returns true if Success, otherwise false. Also re-inits the Fat32 system (call to Fat32_Init). IMPORTANT: -This routine only Quick -->RE<-- formats the card/disk, it should have been initially formatted on a PC (or with the "Fat32_Format" routine), so the MMC/SD/CF card or disk should already contain a valid Fat boot Record. - **Do not forget to re-init the File Variables afterwards**.

Important: "**Fat32_Quickformat**" assumes a valid Fat32 boot record present on the card/disk: ***the card/disk should be formatted first on a PC***.

Example:

```
Fat32_QuickFormat('VolName'); // the volume label will be "VolName"
```

`procedure Fat32_VolumeLabel(var _Label: string);`
Returns the Fat32 Volume Label in "_Label", "Fat32_Init" must have been executed with success before this procedure can be used.

Example:

```
var Label: string[11];
...
Fat32_VolumeLabel(Label);
```

11.2 The Flush routine

`procedure Fat32_Flush(var FileVar: TFileVar);`
Writes the sectorbuffer of the currently assigned file to the card/disk (if necessary), and also writes the (changed) filesize to its directory entry (if necessary). Calling this function writes all info as if the file closes, but keeps the file open for further access.

Example:

```
If Fat32_Assign(File1, 'TestFile.txt', faCreate) then
begin
   // the file is opened successfully, it did not exist it has been created
   // do handle the file content
   Fat32_Flush(File1); // all data is written to the medium as if it was closed,
                       // but the file stays open for processing… The file pointer
                       // position is not altered
end else…
```

11.3 Filesize routines

`function Fat32_Get_File_Size(var FileVar: TFileVar): DWord;`
Returns the FileSize of the currently assigned file in bytes.

`function Fat32_Get_File_Size_Sectors(var FileVar: TFileVar): DWord;`
Returns the FileSize of the currently assigned file in Sectors. A not full last sector is taken into account.

Example:

```
var Size: DWord;
...
Size := Fat32_Get_File_Size(File1)
```

11.4 File date routines

The "Dates" of a file (creation or modified) are not handled automatically by the Fat32_2 library. It is the responsibility of the user to set them correctly (if required).

`procedure Fat32_Get_File_Date(var FileVar: TFileVar; var Year: word; var Month: byte; var Day: byte; var Hours: byte; var Mins: byte);`
Gets the "Creation" date and time of the currently assigned file.

`procedure Fat32_Set_File_Date(var FileVar: TFileVar; Year: word; Month: byte; Day: byte; Hours: byte; Mins: byte);`
Sets the "Creation" date and time of the currently assigned file.

`procedure Fat32_Get_File_Date_Modified(var FileVar: TFileVar; var Year: word; var Month: byte; var Day: byte; var Hours: byte; var Mins: byte);`
Gets the "Last Modified" date and time of the currently assigned file.

`procedure Fat32_Set_File_Date_Modified(var FileVar: TFileVar; Year: word; Month: byte; Day: byte; Hours: byte; Mins: byte);`
Sets the "Last Modified" date and time of the currently assigned file.

Example:

```
Fat32_Set_File_Date(File1, 2009, 9, 14, 20, 51);           // Y, M, D, H, m
Fat32_Set_File_Date_Modified(File1, 2010, 10, 15, 21, 52); // Y, M, D, H, m
```

11.5 File attribute routines

All “attribute” parameters and values are member of this list:

```
  faReadOnly  = $01; // bit 0
  faHidden    = $02; // bit 1
  faSysFile   = $04; // bit 2
  faVolumeId  = $08; // bit 3
  faDirectory = $10; // bit 4
  faArchive   = $20; // bit 5
```

A value of zero means actually “none of the above”, so a “normal” file with the archive bit off.

The Fat32_2 library updates automatically the “Archive” attribute of a file after writing to it. These are the routines to set/reset all attributes and to set/reset the archive attribute separately:

`function Fat32_GetAttr(var FileVar: TFileVar): byte;`
Returns the attributes of the currently assigned file.

`procedure Fat32_SetAttr(var FileVar: TFileVar; Attr: byte);`
Sets the attributes of the currently assigned file.

Example:

```
var Attr: byte;
...
if Fat32_Assign(File1, 'TestFile.txt', faFile) then
begin
  Attr := faArchive + faHidden; // makes the file hidden and to be archived
  Fat32_SetAttr(File1,Attr);
  ...
  Attr := 0;
  Attr := Fat32_GetAttr(File1);
  // check here the current attribute of the file
end;
```

`procedure Fat32_ClearArchiveAttr(var FileVar: TFileVar);`
Clears the archive attribute of the currently assigned file.

`procedure Fat32_SetArchiveAttr(var FileVar: TFileVar);`
Sets the archive attribute of the currently assigned file.

Example:

```
if Fat32_Assign(File1, 'TestFile.txt', faFile) then
begin
  Fat32_SetArchiveAttr(File1); // make the file to be archived
  // or ...
  Fat32_ClearArchiveAttr(File1); // make the file not to be archived
end;
```

11.6 Directory clean up routines

2 special routines to keep the directories themselves clean. There are no routines to defragment non directory files. Cleaning and defragmenting make the creation of new files and “finding” of files and directories faster.

`procedure Fat32_CleanDir(var FileVar: TFileVar);`
"Cleans" the current directory file: deletes the unused entries at the end, which makes it unnecessary to search through them when e.g. testing a file's existence. Enhances speed when creating new files, after other files have been deleted (direntries became free). Attention! !!! this function closes first the currently open file !!!

Example:

```
Fat32_ChDir(File1, 'Directory_One');
Fat32_CleanDir(File1);
```

`procedure Fat32_DefragDir(var FileVar: TFileVar);`
"Defragments" the current directory file: deletes the unused entries "holes" in the directory, which makes it unnecessary to search through them when e.g. testing a file's existence. Enhances speed when creating new files, after other files have been deleted (direntries became free). Does also a "CleanDir". Attention! !!! this function closes first the currently open file !!!

Example:

```
Fat32_ChDir(File1, 'Directory_One');
Fat32_DefragDir(File1);
```

11.7 Storage Size routines

`function Fat32_TotalSpace: DWord;`
Gives the total space in bytes on the Fat32 formatted card. Fat32_Init has to be called first. Only applicable with cards spaces <= 4 GB.

`function Fat32_FreeSpace: DWord;`
Gives the free space in bytes on the Fat32 formatted card. Fat32_Init has to be called first. Only applicable with cards spaces <= 4 GB.

`function Fat32_UsedSpace: DWord;`
Gives the used space in bytes on the Fat32 formatted card. Fat32_Init has to be called first. Only applicable with cards spaces <= 4 GB.

`function Fat32_TotalSpace_KB: DWord;`
Gives the total space in Kilobytes on the Fat32 formatted card. Fat32_Init has to be called first.

`function Fat32_FreeSpace_KB: DWord;`
Gives the free space in Kilobytes on the Fat32 formatted card. Fat32_Init has to be called first.

`function Fat32_UsedSpace_KB: DWord;`
Gives the used space in Kilobytes on the Fat32 formatted card. Fat32_Init has to be called first.

`function Fat32_TotalSpace_MB: DWord;`
Gives the total space in Megabytes (rounded down) on the Fat32 formatted card. Fat32_Init has to be called first.

`function Fat32_FreeSpace_MB: DWord;`
Gives the free space in Megabytes (rounded down) on the Fat32 formatted card. Fat32_Init has to be

called first.

`function Fat32_UsedSpace_MB: DWord;`
Gives the used space in Megabytes (rounded down) on the Fat32 formatted card. Fat32_Init has to be called first.

`function Fat32_TotalSpace_GB: real;`
Gives the total space in Gigabytes on the Fat32 formatted card. Fat32_Init has to be called first.

`function Fat32_FreeSpace_GB: real;`
Gives the free space in Gigabytes on the Fat32 formatted card. Fat32_Init has to be called first.

`function Fat32_UsedSpace_GB: real;`
Gives the used space in Gigabytes on the Fat32 formatted card. Fat32_Init has to be called first.

No examples available for the storage size routines (they speak for themselves).

12 The Fat32_x_MD particularities

In this section the interface and usage of the **Fat32_x_MD** libraries is described. The Fat32_x_MD libraries are capable of handling more than one Fat32 device simultaneously.

12.1 Common

12.1.1 The Fat32 MD Device numbers

The number of Fat32 devices that can be handled is defined by the constant "NrOfFat32Devices" defined in the Fat32_x_MD library. The initial value is 2.

The device numbers themselves (used as "Device" parameter for several functions) ranges from 0 to (NrOfFat32Devices - 1), so initially the device numbers are 0 and 1.

One can of course adapt "NrOfFat32Devices" to the project at hand. After a change re-compilation of the library is required.

The device identified by the Device parameter value in the several functions is defined in the media drivers (see next section).

12.1.2 The Fat32 MD media hardware drivers

Here also at least 2 functions have to be provided by the library user. They have an extra “Device” parameter with respect to the ones needed by the nonFat32_x_MD versions.

Example:

```
{ Device 0 = SDMMC card, Device 1 = USB Stick }

function Fat32_Dev_Read_Sector(Device: byte; Sector: DWord; var Buffer: array[512] of byte):
boolean; // returns true when successful
var Tmp: byte;
begin
  Result := false;
  case Device of
    0: //
    begin
      Tmp := Mmc_Read_Sector(Sector, Buffer);
      Result := (Tmp = 0);
    end;
    1: //
    begin
      Result := USB_MSD_Read_Sector (Sector, Buffer);
    end;
  end;
end;

function Fat32_Dev_Write_Sector(Device: byte; Sector: DWord; var Buffer: array[512] of byte):
boolean; // returns true when successful
var Tmp: byte;
begin
  Result := false;
  case Device of
    0: //
    begin
      Tmp := Mmc_Write_Sector(Sector, Buffer);
      Result := (Tmp = 0);
    end;
    1: //
    begin
      Result := USB_MSD_Write_Sector (Sector, Buffer);
    end;
  end;
end;
```

Additionally, if Fat32_Format is used, the user has to provide the “Fat32_Dev_Capacity_Sectors” function:

```
function Fat32_Dev_Capacity_Sectors(Device: byte): DWord;
var NrOfBlocks: DWord;
    BlockSize: word;
begin
  Result := 0;
  case Device of
  0: begin
       Result := SDMMC_CardSize_Sectors;
     end;
  1: begin
       USB_MSD_Device_Capacity(NrOfBlocks, BlockSize);
```

```
      Result := NrOfBlocks;
    end;
  end;
end;
```

12.1.3 Using more than one SDMMC on the same SPI bus

Since there is only one Chip enable output for MMC cards the follwing has to be done:

Per SDMMC card an additional OR gate must be provided, “oring” the chip select output from the MMC library with extra chip selects (one per SDMMC) set in the media hardware drivers.
The output of the “or” functions is then to be connected to the actual “not CE” input line of the SDMM Cards.

12.2 Fat32_1_MD

In this type of library all routines have an extra parameter called “Device” as the first (or only) parameter.
Examples:

```
function Fat32_Init(Device: byte): boolean;
function Fat32_QuickFormat(Device: byte; var VolumeLabel: string[11]): boolean;
function Fat32_Assign(Device: byte; var LongFn: TLongFileName; file_cre_attr: byte): boolean;
```

etc...

If one wants to init the USB stick using the drivers in section 12.1.2 then it is done by:

```
Fat32_init(1); // device 1 is the USB stick
```

12.3 Fat32_2_MD

In this library, the situation is different, because “File Variables” (of type TFileVar) are used.
The “Device” to which every File Variable is assigned is incorporated in the File Variable.

This means the following:

- The initialisation function for File Variables has one more parameter: the “Device” number.

  ```
  procedure Fat32_FileVar_Init(var FileVar: TFilevar; Device : byte);
  ```

  So, if one wants to use File1 (TFileVar) for the USB stick then this is the way to do it:

  ```
  Fat32_FileVar_Init(File1, 1);// device 1 is the USB stick
  ```

- All routines in the library which do not have a “TFileVar” as first parameter (the “device” related procedures) have an extra first “Device” parameter. e.g.:

  Examples are:

  ```
  function Fat32_Init(Device: byte): boolean;
  function Fat32_QuickFormat(Device: byte; var VolumeLabel: string[11]): boolean;
  ```

```
procedure Fat32_VolumeLabel(Device: byte; var _Label: string);
```

For usage examples see above section 12.2.

- Procedures that had already a “TFileVar” parameter are the same as in the Fat32_2 (non MD) version.

[end of document]