Embedding it better...

µTasker Document

µTasker – utFAT2.1

Version in progress – non-official release!

Table of Contents

1. Introduction

µtFAT V2.1 is a FAT12/16/32/exFAT compatible file system for use with V2 SD cards [*Secure Digital*] and SDHC cards [*Secure Digital High Capacity*] – and also USB memory sticks. It can use the SPI mode of SD cards to allow any supported processor with an SPI interface to be used to read and write data, whereby a single file can be up to 4 GByte in size [16 exabytes for exFAT] and the total storage space from typically 2 GByte to 32 GByte, but up to 2TByte possible (with 512 byte sector size). *When the supported processor has an SDIO (SDHC) controller this is generally supported too as a more efficient method of data transfer*.

The user interface is designed to be practical and comfortable (encapsulating typically required detail work like displaying directory contents). The code is designed for minimum RAM requirements (from about 600 Bytes for a single user interface).

µtFAT is (optionally) fully integrated into the µTasker HTTP and FTP servers and a user interface (DOS-like) is included in the µTasker project for simple test and study. Furthermore µtFAT and SD cards/memory sticks can be used and tested within the µTasker simulator, allowing comfortable project development and testing as well as greatly simplified study of the software.

Optionally, µtFAT can read and write long file names (*LFN*) – exFAT always uses a LFN technique. It can optionally format disks, create, rename and delete directories and files, plus write file content.

µtFAT neither supports V1 SD cards nor MMCs. Due to the fact that it is becoming increasingly difficult to purchase SD cards rather than SDHC cards the smaller, older ones are considered legacy devices - µtFAT V2.0 arrived at the time when it made sense to concentrate fully on present-day technology.

A single SD card is supported (it is generally referenced as disk `D:\`) with either no partition or a single partition. The module is however prepared for extension to multiple partitions if this proves to make sense during further development. A memory stick is generally references as disk `E:\`).

Since SD cards (and memory sticks) are removable the µtFAT module includes automated support for detecting and mounting them so that the user doesn't need to be involved with these details. When the SD card/memory stick is not detected the internal file system can fall back to the µFileSystem so that the web server, for example, can still display information about the fact that there is no media present and can still allow embedded applications to operate using the fall-back interface if required.

The µtFAT module thus incorporates the following elements:

- SD card interface in SPI mode and/or SDIO (SDHC) controller mode (when supported by the processor in use)
- Full speed or high speed USB host to memory stick (when available in the processor)
- FAT12/16/32 with optional long file name (LFN) support. When LFN writing is enabled it can use full LFN compatibility or Linux work-around mode to avoid patent issues
- ExFAT with its larger file size capabilities
- Memory management interface controlling automated detection and mounting of the SD card/memory stick and enabling supervision of file protection and sharing
- Simplified user interface incorporating a variety of standard tasks like listing directories
- Optional data cache on a per-file basis to optimise access speed in case of small data reads/writes

- Full integration into µTasker HTTP and FTP servers with fall-back capability to µFileSystem in internal FLASH or external SPI FLASH
- Optional utFAT expert mode allowing analysis of file object storage, corruptions and deleted content
- Optional safe deletion and undelete functions

The demo project include a DOS-like interface via UART, USB CDC or TELNET allowing management of files and directories as well as study of the µtFAT module and FAT16/32 in general.

This document has the following goals:

- To present the µtFAT user interface so that it can be effectively used in µTasker projects
- To discuss the relevant details of SD cards operation when used in µTasker projects with the µtFAT
- To discuss the relevant details of FAT12/16/32/exFAT so that users of µtFAT and the µTasker project fully understand its underlying operation. In addition to serve as a study aid when learning about FAT12/16/32/exFAT operation, especially when working together with the µTasker simulator and its SD card/memory stick simulation capabilities
- To discuss the relevant details about file management (sharing and protection) in a multi-user embedded system with real-time demands
- To highlight implementation details to fulfil long file name write operations

The following references serve as basis for SD card and FAT12/16/32 specifications:

- SD-Association – Part 1 - Physical Layer Simplified Specification: https://www.sdcard.org/downloads/pls/
- Microsoft Extensible Firmware Initiative FAT32 File System Specification: http://www.microsoft.com/whdc/system/platform/firmware/fatgen.mspx

The exFAT specification is available at
https://learn.microsoft.com/en-us/windows/win32/fileio/exfat-specification

There are many additional sources for information available in the Internet so this document restricts details to those of particular relevance to the use of the µtFAT module.

At the time of writing the µtFAT module has been tested together with the µTasker demo project on the following processors (on various evaluation and demo boards):

- ATMEL AT91SAM7X
- ATMEL AVR32 UC3A, UC3B and UC3C
- Freescale M522XX and Kinetis KE, KL, K (SPI and SDHC)
- Luminary Micro (TI) LM3Sxxxx
- NXP LPC2XXX (SPI and SDIO) and LPC17XX
- ST STR91XF and STM32 (SPI and SDIO)
- i.MX RT 10xx SPI and SDHC
- Kinetis FS and HS USB host to memory stick
- i.MX RT 10xx HS USB host to memory stick

A complete project including full options (command line interface via USB, UART and TELNET with DOS-like µtFAT menu), TCP/IP with HTTP, FTP servers, TELNET, etc. and USB CDC occupies around 70k code space and 30k RAM on an ARM processor, whereby the µtFAT module contributes around 10k code and 1k RAM. The complete project on a Coldfire occupies, in comparison, about 100k code space and 30k RAM whereby the µtFAT module contributes about 16k code and 1k RAM.

The reference project supports formatting SD cards/memory sticks, copying data to the storage media via FTP, serving web server content up to the capacity of the SD card/memory stick as well as dynamic content generation, fall back to the µFileSystem when no SD card/memory stick is inserted as well as the complete functionality of the µTasker V2.0 project.

This document uses FAT32 as basis for discussions. FAT16/FAT12 are optionally supported and details that vary in comparison with FAT32 are given mainly as foot-notes to the FAT32 descriptions.

FAT16 support can be optionally enabled with the define

```
#define UTFAT16
```

FAT12 support can be optionally enabled with the define

```
#define UTFAT12
```

exFAT32 is optional and is discussed in a chapter of its own, whereby the main differences to FAT32 are highlighted there. It can be enabled with the defined

```
#define UTEXFAT
```

The full list of project defines are:

```
#define SDCARD_SUPPORT              // enables SD card with utFAT
#define SD_CARD_RETRY_INTERVAL 5    // attempt SD card initialisation at 5s
                                       intervals
#define UT_DIRECTORIES_AVAILABLE 5  // this many directories objects are available
                                       for allocation
#define UTMANAGED_FILE_COUNT  10    // allow this many "managed files" at one time
#define UTFAT_LFN_READ              // enable long file name read support
#define MAX_UTFAT_FILE_NAME   (100) // the maximum file name length supported (LFN)
                                       - maximum 255
#define UTFAT_WRITE                 // enable write functions
```

```
#define UTFAT_FORMATTING            // enable formatting SD cards (requires also
                                       UTFAT_WRITE)
#define UTFAT_FULL_FORMATTING       // enable formatting SD cards including zeroing
                                       of data sectors as well as FAT sectors
#define UTFAT_LFN_DELETE            // support deleting files with LFN
                                       (cleaning up all LFN directory entries) when
                                       LFN write is not enabled
#define UTFAT_LFN_WRITE             // enable LFN write functions
#define UTFAT_LFN_WRITE_PATCH       // enable LFN write functions based on Linux
                                       patch to potentially avoid possible patent
                                       issues
#define SFN_ENTRY_CACHE_SIZE  20    // short file name cache used to speed up SFN
                                       alias collision searching when writing LFNs
#define UTFAT_SAFE_DELETE           // delete operation removes all information so
                                       that no undelete is possible
#define UTFAT_UNDELETE              // undelete support for files and directories
#define UTFAT16                     // support FAT16 as well as FAT32
#define UTFAT12                     // support FAT12 as well as FAT32
#define UTEXFAT                     // support exFAT as well as FAT32
#define UTFAT_FILE_CACHE_POOL 2     // file data cache buffers in the pool
#define UTFAT_EXPERT_FUNCTIONS      // enable additional functions for monitoring
                                       operation and performing advanced operations
#define SUPPORT_FILE_TIME_STAMP     // when activated, fnGetLocalFileTime() must
                                       exist, which return the date and time
                                       information
#define SD_CONTROLLER_AVAILABLE     // set when SDIO/SDHC controller available and
                                       is to be used instead of SPI
#define UTFAT_SECT_BIG_ENDIAN       // display sector content as big-endian view on
                                       little-endian processors or when simulating
#define UTFAT_SECT_LITTLE_ENDIAN    // display sector content as little-endian view
                                       on big-endian processors

#define USB_INTERFACE               // enable USB interface
#define USB_HOST_SUPPORT            // enable USB host stack
#define USB_MSD_HOST                // support MSD host (memory stick)
```

Note that the three USB defines should all be defined in order enable memory stick support. SD card and memory sticks can both be used at the same time, as can multiple USB memory sticks when the microprocessor supports multiple USB host interfaces.

Important: Please read the disclaimers at the end of this document. The use of the µtFAT module implies their acceptance in their entirety.

2. Electrical Connection to the SD card in SPI Mode

A standard SD card measures 32 mm × 24 mm × 2.1 mm and has 9 pins as show in figure 2-1.

Figure 2-1 Standard SD card pin numbering

The pins use in SPI mode are detailed in table 2-1

Pin	Name	Direction	Description
1	CS	Input	Chip Select
2	DI	Input (push-pull)	Data In
3	VSS	-	Supply voltage ground
4	VDD	Power input	Supply voltage
5	SCLK	Input	Clock
6	VSS2	-	Supply voltage ground
7	DO	Bi-directional (push-pull)	Data Out
8		RSV	
9		RSV	

Table 2-1 SD card pin descriptions in SPI mode

Connecting an SD card to any processor with an SPI interface is very easy since it involves connecting the standard SPI signals (MISO, MOSI and SPCLK) and a single chip select line along with power of typically 3V3. Some designs will allow the processor to turn on and off the power to the SD card so that it can be powered down during insertion and removal and to save power consumption when not used. Due to the design of the contacts to the SD card it is however generally not a problem to insert and remove with power applied (hot-plugging).

The communication and control signals are thus simply 4 wires as follows:

- Pin 1 – CS – CS-line (output processor port asserted ‘0’ to enable the SD card)
- Pin 2 – DI – MOSI (from the SPI interface of the processor)
- Pin 4 – SCLK – SPCLK (from the SPI interface of the processor)
- Pin 7 – DO – MISO (from the SPI interface of the processor)

A further popular format is the microSD/microSDHC format which measures only 15 mm x 11mm x 1mm. Its popularity is mainly due to its acceptance by mobile handset vendors because of its ultra-compact size and widely supported standard SD interface. These can also be used in SD card sockets together with an adapter. They have only 8 pins as shown in SPI mode in table 2-2

Pin	Name	Direction	Description
1	NC	-	No contact
2	CS	Input	Chip select
3	DI	Input (push-pull)	Data In
4	VDD	Power input	Supply voltage
5	SCLK	Input	Clock
6	VSS	-	Supply voltage ground
7	DO	Bi-directional (push-pull)	Data Out
8		RSV	

Table 2-2 microSD card pin descriptions

The communication and control signals are thus simply 4 wires as follows:

- Pin 2 – CS – CS-line (output processor port asserted ‘0’ to enable the SD card)
- Pin 3 – DI – MOSI (from the SPI interface of the processor)
- Pin 5 – SCLK – SPCLK (from the SPI interface of the processor)
- Pin 7 – DO – MISO (from the SPI interface of the processor)

3. SPI Mode of Operation

Since almost every processor has an SPI interface the SPI mode is the most important mode for general embedded operation. SD-modes of operation include (in addition to SPI mode) 1-bit and 4-bit modes, which involves four data lines and requires the processor to have an SD/MMC interface; this allows greater data throughput to be obtained but is not generally necessary.

It is possible that not all SDHC cards will support the SPI mode in the future, which may restrict the SPI use on higher capacity devices, however the SPI interface is not expected to generally die out since it certainly enables the simplest and cheapest interface using general purpose processors, which will continue to be of importance also in the years to come.

The SPI mode is not the default mode of operation and must be forced by applying the following procedure:

- Once the SD card is powered (at least 1ms after its input voltage reaches 2V2) the DI and CS lines are held high and at least 74 clock pulses (100kHz to 400kHz) are applied to SCLK.
- The SD card has now entered its *native command mode*.
- With the clock speed still set between 100k and 400k a software reset is commanded using the `GO_IDLE_STATE` (CMD0) command (with valid CRC). The command is issued with the CS line low, causing the card to enter *SPI mode*.

The `GO_IDLE_STATE` is an example of an SD card command – the complete set of commands is included in the *SD-Association – Part 1 - Physical Layer Simplified Specification*. Not all commands are required in order to realise the interface.

To issue this command (generally true for all SD card commands) the following procedure is used:

1) The SPI bus is read to ensure that the card is ready to receive commands. This is indicated by the value 0xff being read from the bus (note that if no SD card is inserted the bus state may be detected as 0xff or 0x00 depending on whether the DI line is pulled up or not)
2) If the SPI bus is not reading 0xff the card may be busy so the driver software is required to allow the SD card some time to complete and thus poll until detected as ready
3) SD card commands are 6 bytes in length, whereby the `GO_IDLE_STATE` consists of the fixed content `0x40, 0x00, 0x00, 0x00, 0x00, 0x95`. The first byte is the command; the command is also known as CMD0 (the actual command value is equal to 0x40 + CMD number) and the final byte (0x95) is a checksum over the command length. The `GO_IDLE_STATE` command has not parameters and the 4 x 0x00 are stuff-bytes. *Since the command has always the same content its checksum is always the same*
4) After transmission of the 6 byte command the result is read from the SD card. It is also possible (depending on command type) that the SD card requires some time to complete the command and it will indicate this by keeping the most significant bit of the result value (0x80) at '1' (busy). The driver software should wait until the SD card no longer indicates that it is busy (by polling the result value) before it can return the actual result from the executed command

5) The `GO_IDLE_STATE` command will return the SD card's state which is intially expected to be the IDLE state (value 0x01). Some commands return a result plus extra information ,which is read by subsequently read bytes; *for example, the command 58 returns the Operations Condition Register (OCR), containing 4 additional bytes of data*

The SPI mode must be set so that the clock and data have the format as shown in figure 3-1. This is 8-bit MOTOROLA mode with clock phase set so that the MOSI data changes on the falling edge of the SPCLK:

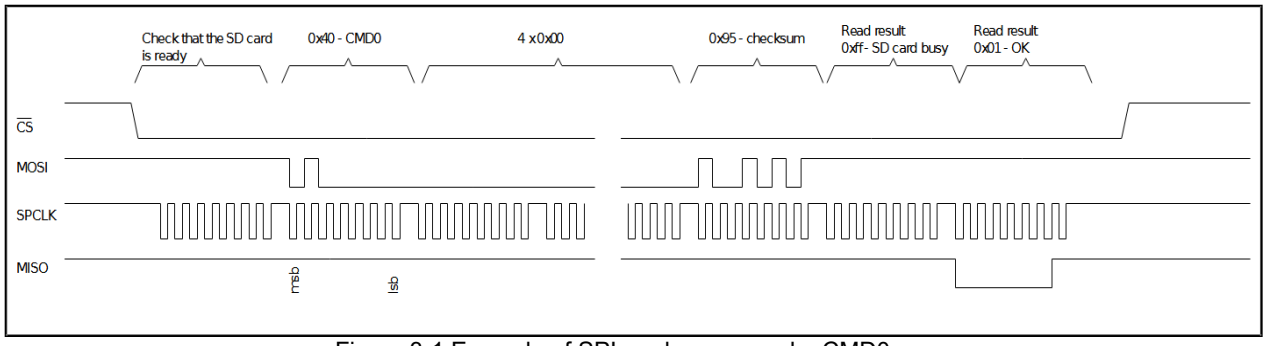


Figure 3-1 Example of SPI mode command – CMD0

4. SD Mode of Operation

The pin-out of the SD card in SD (4-bit) mode is detailed in table 4-1.

Pin	Name	Direction	Description
1	CD/DAT3	Bi-directional (push-pull)	Card Detect/Data 3
2	CMD	Input	Command in
3	VSS	-	Supply voltage ground
4	VDD	Power input	Supply voltage
5	CLK	Input	Clock
6	VSS2	-	Supply voltage ground
7	DAT0	Bi-directional (push-pull)	Data 0
8	DAT1	Bi-directional (push-pull)	Data 1
9	DAT2	Bi-directional (push-pull)	Data 2

Table 4-1 SD card pin descriptions in SD mode

The pin-out of the microSD card in SD (4-bit) mode is detailed in table 4-2.

Pin	Name	Direction	Description
1	DAT2	Bi-directional (push-pull)	Data 2
2	CD/DAT3	Bi-directional (push-pull)	Card Detect/Data 3
3	CMD	Input	Command in
4	VDD	Power input	Supply voltage
5	CLK	Input	Clock
6	VSS	-	Supply voltage ground
7	DAT0	Bi-directional (push-pull)	Data 0
8	DAT1	Bi-directional (push-pull)	Data 1

Table 4-2 microSD card pin descriptions in SD mode

The most important difference between SPI mode and SD mode is that the SD mode supports 4-bit data mode. The initialisation phase operates in 1-bit mode and the bus width is commanded to 4-bit during the initialisation sequence.

5. SD Card Initialisation Phase

The initialization of the SD card is managed by the mass-storage task. This task is activated in the µTasker project from the application task using the command:

```
uTaskerStateChange(TASK_MASS_STORAGE, UTASKER_ACTIVATE);
```

The mass storage task realises a state-event machine which will attempt to initiate the SD card. If the initialization fails (probably due to the SD card not being inserted) it will retry at regular intervals as defined by:

```
#define SD_CARD_RETRY_INTERVAL 5  // attempt SD card initialisation at 5s intervals
```

The initialization involves forcing the SD card to the SPI mode (if SPI is used) as described in the previous section, followed by reading and checking the SD card type and attributes – at least a V2 SD card is expected for the card to be accepted for further operation. Typical sequences are described in the *SD-Association – Part 1 - Physical Layer Simplified Specification*.

During the initialisation sequence the state-event machine also allows cards to respond slowly by interrupting polling to allow other tasks in the system to be scheduled. This is also important when a card is not present and enough time is being given before declaring the initialisation attempt as failed. The process thus behaves as a background activity and doesn't impact general system operation even when the initialisation is attempted at regular intervals.

Once all information about the card has been retrieved and checked for validity the SPI interface speed is increased from the initial 100..400kHz to the higher operating speed of up to 25MHz. *The speed increase is also valid for SDIO (SDHC) mode, whereby the data bus width is also changed for the default 1-bit mode to 4-bit mode*.

6. SD Card Disk Mounting

Once the SD card initialisation has completed the mounting phase begins. This is also performed automatically by the mass storage task's state-event machine as a background activity.

It starts by reading the first sector on the SD card in an attempt to identify an existing FAT formatted card. This sector is known as the boot sector, containing a BIOS parameter block, but it may turn out that it is not such a sector but instead is an extended boot record supplying information about partitions on the disk – *this detail is in fact missing from the Microsoft FAT32 File System Specification*.

Before looking at the content of such sections it is worth noting that an SD card section is accessed by first issuing the SD card command 17 (`READ_SINGLE_BLOCK_CMD17`) followed by reading 512 bytes of data from the specified sector. The sector size is always 512 bytes and this is the block size used when reading and writing. The command `READ_SINGLE_BLOCK_CMD17` includes also a parameter containing the sector which is to be read, which can be a sector number (SDHC cards) or a byte offset (SD cards) – this detail is controlled in the driver based on information concerning the type of SD card being used which was obtained during the initialisation phase.

The 512 bytes read can then be interpreted. Both an extended boot record and a BIOS parameter block contain a check pattern of 0x55 and 0xaa as last two bytes of the sector, which is used as a first simple check that the sector's content is valid. If this is not the case the SD card is not formatted in any recognisable way and so cannot be mounted. It must then either be formatted in a formatting device (like in an SD card slot in a PC) or by commanding the formatting if the µtFAT formatting support option is enabled.

The two possible valid sector content types are compared below.

```
typedef struct _PACK stEXTENDED_BOOT_RECORD
{
    unsigned char         EBR_unused1[394];         // generally 0
    unsigned char         EBR_IBM_menu[9];          // possible IBM boot manager menu entry
    unsigned char         EBR_unused2[43];          // generally 0
    PARTITION_TABLE_ENTRY EBR_partition_table[2];   // two partition tables
    unsigned char         EBR_unused3[32];          // generally 0
    unsigned char         ucCheck55;                // this location must be 0x55 - offset 510
    unsigned char         ucCheckAA;                // this location must be 0xaa - offset 511
} EXTENDED_BOOT_RECORD;
```

Code 6-1 Sector content when extended boot record

where each partition table entry is:

```
typedef struct stPARTITION_TABLE_ENTRY
{
    unsigned char boot_indicator;                   // 0x80 indicates bootable
    unsigned char starting_cylinder;                // cylinder start value
    unsigned char starting_head;                    // head start value
    unsigned char starting_sector;                  // sector start value
    unsigned char partition_type;                   // partition type descriptor
    unsigned char ending_cylinder;                  // cylinder start value
    unsigned char ending_head;                      // head start value
    unsigned char ending_sector;                    // sector start value
    unsigned char start_sector[4];                  // start sector
    unsigned char partition_size[4];                // partition size in sectors
} PARTITION_TABLE_ENTRY;
```

Code 6-2 Partition entry table content

```
typedef struct _PACK stBOOT_SECTOR_FAT32
{
    BOOT_SECTOR_BPB boot_sector_bpb;   // standard boot sector and bios parameter block
    unsigned char BPB_FATSz32[4];      // FAT32 32-bit count of sectors occupied by ONE FAT -
                                          BPB_FATSz16 must be zero!
    unsigned char BPB_ExtFlags[2];
    unsigned char BPB_FSVer[2];        // version number of the FAT32 volume. major:minor -
                                          0:0 expected at the time of writing but could
                                          change in the future indicating changes
    unsigned char BPB_RootClus[4];     // cluster number of the first cluster of the root
                                          directory. Usually 2
    unsigned char BPB_FSInfo[2];       // sector number of FSINFO structure in the reserved
                                          area of the FAT32 volume. Usually 1
    unsigned char BPB_BkBootSec[2];    // sector number in the reserved area of the volume of
                                          a copy of the boot record (if non-zero). 6 is
                                          recommended
    unsigned char BPB_Reserved[12];    // should be 0
    unsigned char BS_DrvNum;           // int 0x13 drive number - operating system specific
    unsigned char BS_Reserved1;        // should always be set to zero when formatting (is in
                                          fact used by Windows NT)
    unsigned char BS_BootSig;          // extended boot signature (0x29). Signature indicating
                                          that the following three fields are present
    unsigned char BS_VolID[4];         // volume serial number. Usually generated by simply
                                          combining the current date and time into a 32-bit
                                          value
    CHAR          BS_VolLab[11];       // label matching the 11-byte volume label recorded in
                                          the root directory. "NO NAME    " when no specific
                                          label
    CHAR          BS_FilSysType[8];    // always "FAT32   ". Not actually used to determine
                                          type (more informational) and not used at all by
                                          Microsoft FAT
    unsigned char ucSpace[420];
    unsigned char ucCheck55;           // this location must be 0x55 - offset 510
    unsigned char ucCheckAA;           // this location must be 0xaa - offset 511
} BOOT_SECTOR_FAT32;
```

Code 6-3 Sector content when FAT32 boot sector

where the boot sector and BIOS parameter block content is:

```
typedef struct stBOOT_SECTOR_BPB       // boot sector and bios parameter block
{
    unsigned char BS_jmpBoot[3];       // jump instruction to boot code
    CHAR          BS_OEMName[8];       // string usually indicating the system that formatted
                                          the volume - "MSWIN4.1" is recommended although
                                          MSDOS5.0 is typical
    unsigned char BPB_BytesPerSec[2];  // count of bytes per sector. This value may take on
                                          only the following values: 512, 1024, 2048 or 4096
    unsigned char BPB_SecPerClus;      // number of sectors per allocation unit. The legal
                                          values are 1, 2, 4, 8, 16, 32, 64, and 128 - however
                                          never cause a "bytes per cluster" value
                                         (BPB_BytesPerSec * BPB_SecPerClus) greater than 32K!!
    unsigned char BPB_RsvdSecCnt[2];   // number of reserved sectors in the reserved region of
                                          the volume starting at the first sector of the
                                          volume. Never 0. FAT12/16 always 1. FAT32 uses
                                          typcially 32
    unsigned char BPB_NumFATs;         // the count of FAT data structures on the volume.
                                          Recommended to be always 2 (although FLASH could use
                                          1)
    unsigned char BPB_RootEntCnt[2];   // FAT12 and FAT16 volumes count of 32-byte directory
                                          entries in the root directory. FAT32 must always be
                                          0. FAT16 should use 512. When multiplied by 32 it
                                          should result in an even multiple of BPB_BytesPerSec
                                          (for FAT12 and FAT16)
    unsigned char BPB_TotSec16[2];     // old 16-bit total count of sectors on the volume (in
                                          all four regions of the volume). May be zero is
                                          BPB_TotSec32  is non-zero. Must be 0 for FAT32
    unsigned char BPB_Media;           // 0xF0, 0xF8, 0xF9, 0xFA, 0xFB, 0xFC, 0xFD, 0xFE, or
                                          0xFF. 0xF8 usually used for fixed and 0xF0 for
                                          removable media. Should match with FAT[0] entry but
                                          is otherwise obsolete
```

```
    unsigned char BPB_FATSz16[2];      // FAT12/FAT16 16-bit count of sectors occupied by ONE
                                          FAT. Must be 0 for FAT32
    unsigned char BPB_SecPerTrk[2];    // sectors per track for interrupt 0x13. Only valid for
                                          media whose volume is broken down into tracks by
                                          multiple heads and cylinders
    unsigned char BPB_NumHeads[2];     // number of heads for interrupt 0x13.
    unsigned char BPB_HiddSec[4];      // count of hidden sectors preceding the partition that
                                          contains this FAT volume. Should always be zero on
                                          media that are not partitioned but otherwise
                                          operating system dependent
    unsigned char BPB_TotSec32[4];     // 32-bit total count of sectors on the volume (all
                                          sectors in all four regions). Can be zero if
                                          BPB_TotSec16 is non-zero. Must be non-zero for FAT32
} BOOT_SECTOR_BPB;
```

Code 6-4 Boot sector and BIOS parameter block

It is fairly easy to recognise an extended boot record because there is partition information available where a FAT32 boot sector would normally have zeros. In addition, the FAT32 boot sector would have the string FAT32 at the location `BS_FilSysType`.

The µtFAT V2.01will use just use the first partition if one is found. The partition entry parameter `start_sector[4]` indicates the location of the boot sector to be used. FAT32 uses little-endian format for storage so the sector to be read to load the boot sector (which would already be loaded if there were no extended boot record used) is given by:

```
ulSector = ((start_sector[3] << 24) + (start_sector[2] << 16) +
            (start_sector[1] << 8) + start_sector[0]);
```

in order to ensure that the long word value is correct irrespective of the processor architecture being used.

In later code bases the method

```
ulSector = fnSafeGetBufLittleLong(start_sector);
```

will typically be found, using a routine that performs the correct extraction and interpretation and also ensures that optimisers will never use unaligned accesses in the process.

After reading this sector (remembering that a sector is always 512 bytes in length), the FAT32 boot sector can be interpreted as the mounting process is continued. The content of the FAT32 boot sector is interpreted to define various parameters to verify that it is indeed FAT32 formatted and various details are retained for later operational use. These details are not very complicated but there are quite a lot of values; some of which were important for floppy drives but no longer of much relevance, and others which are essential for correct operation later. It takes some experience with the values before they start making much sense and it is not the objective of this document to explain what they all mean and exactly how each one can be used, or ignored depending on their significance. Instead, rather than getting bogged down with details it is a good point to move on to a more practical study approach which should help in understanding how this encrypted information is put to use during FAT32 operation.

Therefore it is adequate to state a few small details for the moment so that the next stage can already begin:

1) If the cluster count is smaller than 65525 it is not FAT32 and is *probably* FAT16 or FAT12. It is possible that some SD cards are formatted per default as FAT16

although they could be formatted as FAT32 – also Windows may format a 2G SD cards as FAT16 if the FAT32 option is not specifically set. Such SD-cards can however be reformatted accordingly.

2) Once all information has been collected from the FAT32 boot section and is determined as valid the mounting phase is complete. From this point on the SD card can be used.

3) To find out important details about the SD card and its FAT configuration the "`Disk Interface`" menu of the µTasker demo project can be used. In addition, the contents of SD card sectors can be displayed so that the internal workings soon become quite clear.

4) The µtFAT V2.1 includes an SD card simulator. The simulator can also be used to perform the same tests as with a real SD card on target hardware with the advantage that it is faster (reformatting a real 2G SD card may take several minutes, but the simulator allows it to be tested in about 2 seconds), doesn't involve modifying real content and also allows comfortable debugging (code stepping) for anyone interested in the internal workings of the µtFAT module, including SPI or SDIO (SDHC) driver.
The size of the SD card used when simulating can be defined by the define
```
#define SDCARD_SIM_SIZE   SDCARD_SIZE_2G
```
This would cause the simulator to work as a virtual 2 GByte card but sizes of 1G, 2G, 4G, 8G and 16G are supported.
The content of the simulated SD card is saved in the simulator directory as `SD_CARD.bin`. Changes are immediately made in this file and it is not stored only on exit from the simulator as is internal flash.
If no SD card size is specified the SD card simulator defaults to a 1GByte card.
Should the user want to simulate the project without an SD card inserted initially the define
```
#define _NO_SD_CARD_INSERTED
```
can be used. A simulator card can be inserted later by clicking on the SD card symbol in the simulator.

5) The µtFAT V2.1 includes a memory stick simulator when USB host is enabled and either `SIMULATE_MEMORY_STICK_WITHOUT_STALL` or `SIMULATE_MEMORY_STICK_WITH_STALL` is enabled. The size of the stick is defined by
```
#define EMULATED_MEMORY_STICK_SIZE (128043712000/512)
 // emulate 128GByte drive
```
The memory stick can be inserted by using the simulator's USB menu and the command "Attach full speed device" or "Attach high speed device" which allows the emulated stick to enumerate and subsequently be used for FAT operations in a similar manner to the SD card emulation.

The stick can be removed (ad re-inserted later if required) by using the USB menu command "Disconnect".

Note that processors with two USB controllers (like the Kinetis K66 or i.MX RT 10xx) will default to using the first controller. If the commands are to be applied to the second controller the second USB interface can be selected by clicking on it as shows in this video: https://youtu.be/3irORtbc4e0

The content of the simulated memory stick is saved in the simulator directory as `MEMSTICK0.bin` (or `MEMSTICK1.bin` if connected to a second USB controller).

7. First Steps with an SD Card and Understanding FAT32

An SD card can be in one of 5 possible states

- *Not present* – in this case it is not usable until inserted
- *Inserted but not formatted* – in this case it cannot yet be used and must first be formatted
- *Mal-formatted* – this could be due to an error or because its formatting is neither FAT32 nor FAT16 not FAT12 not exFAT (when the FAT options are enabled). In this case it needs to be reformatted or the corresponding FAT option enabled in order to work with it in that format
- *FAT formatted and completely empty* – in this case it is in a fresh state with no directories or files and also no traces of old directories and files (which can often be recovered quite easily – eg. undeleted)
- *FAT formatted with data* – in this case it contains directories and/or files and potentially also traces of deleted directories and files

We will start with an inserted but not formatted 2 GByte SD card and work through formatting it and then using it to store data on. This can be performed by starting with a non-formatted card on a target board loaded with the µTasker demo project including the µtFAT module or else by running the µTasker simulator. Apart from some possible differences in reaction time the results should be identical; the following assumes that the µTasker simulator is used and gives some extra details concerning this where necessary – the simulator details can be ignored if only target operation is of interest.

This is where things get practical and the rather brief but intensive details of the last sections should quickly fall into place so that the module starts to become rather more fun!

7.1. Checking the Details of a Non-Formatted SD Card

To do this a non-formatted SD card is required. If no such card is available it is not a problem since this step is rather academic and can be skipped if needed.

When using the simulator a non-formatted SD card can simply be created by ensuring that the file `SD_CARD.bin` in the simulation directory `\Applications\uTaskerV1.4\Simulator` either doesn't exist or is deleted. When the µtFAT module runs in the µTasker simulator environment it is this file which is used to store all SD card data in – when it is empty there is no data and also no boot sector information.

Connect to the menu interface either via UART or TELNET (USB CDC can also be used on a target but UART or TELNET are more suitable for simulator operation). Now enter the "SD card disk interface" where the following commands are displayed (depending on project options some additional commands may also be displayed):

```
  Disk interface
==================
up            go to main menu
info          utFAT/card info
dir           [path] show directory content
cd            [path] change dir. (.. for up)
file          [path] new empty file
write         [path] test write to file
mkdir         new empty dir
rename        [from] [to] rename
print         [path] print file content
del           [path] delete file or dir.
format        [-16/12/ex] [label] format (unformatted) disk
re-format     [-16/12/ex] [label] reformat disk!!!!!
sect          [hex number] display sector
help          Display menu specific help
quit          Leave command mode
```

Terminal 7-1 Disk Interface Menu

Since there is no formatted SD card available any attempt to perform file system operations, like "`DIR`" will result in a message "`No SD-Card ready`". As long as an SD card is however inserted (which can be set as the default case when simulating) its information will have been read and so the "`info`" command will work as below:

```
SD-card not formatted (1977614336 bytes)
CSD: 0x00 0x26 0x00 0x32 0x5f 0x5a 0x83 0xae 0xfe 0xfb 0xcf 0xff 0x92 0x80 0xff 0x00
```

Terminal 7-2 "info" command response from an unformatted SD card

This is showing that the SD card is not formatted but has about 2 GByte of usable space. In addition, its CSD (Card Specific Data) register is displayed. This contains various details about the card including its space, speed etc. - see chapter 5.3 of the `SD-Association – Part 1 – Physical Layer Simplified Specification` for complete details concerning interpreting these values.

7.2. Formatting or Re-formatting an SD Card

A non-formatted SD card can be formatted in a PC but, as long as the options `UTFAT_WRITE` and `UTFAT_FORMATTING` are set, it can also be formatted by the µtFAT module. To do this the command "`format`" is used; to re-format a formatted SD card the command "`re-format`" achieves the same result. ***WARNING: content will be lost when this command is used – that is the reason why the command "re-format", with hyphen, was chosen!!***

Fully formatting a real SD card can take several minutes since it involves writing a large amount of data space – when simulating it takes about 2 seconds. The formatting takes place by writing the partition information and boot sector information accordingly, as well as resetting all FAT content. In addition, a volume label can be passed with the formatting command (up to 11 ASCII characters in length*) which will be displayed when the SD card is inserted into a PC. The following shows the formatting command and the response to a subsequent "`info`" command, where the volume ID is also visible:

```
>format UTFAT

Formatting in progress - please wait...
>**Disk D formatted
Disk D mounted
info


SD-card UTFAT (1977614336 bytes)
Bytes per sector: 512
Cluster size: 4096
Directory base: 0x00000002
FAT start: 0x0000005f
FAT size: 0x00000eb9
Number of FATs: 2
LBA: 0x00001dd1
Total clusters: 0x00075a45
Info sect: 0x00000040
Free clusters: 0x00075a44
Next free: 0x00000003
CSD: 0x00 0x26 0x00 0x32 0x5f 0x5a 0x83 0xae 0xfe 0xfb 0xcf 0xff 0x92 0x80 0xff 0x00
>
```

Terminal 7-3 "info" command response from a formatted SD card

**exFAT formatted disks can have volume labels of up to 22 characters (unicode is also supported).*

Note that the formatting process is controlled by the mass-storage task in a manner that it can operate as a background process during normal system operation.

- If the project option

  `#define UTFAT_FULL_FORMATTING`

  Is enabled additional commands are available for formatting and reformatting card: `fformat` and `re-fformat`
  The use of these commands is the same as the `format` and `re-format` commands but they stand for 'full' format and not only delete the FAT but also all content of the cluster area to ensure that the complete SD card content is deleted. This process can however take a long time to complete on a large SD card!

- If FAT16 is enabled, a card can be formatted with FAT16 by using the optional `[-16]` flag `format -16 [optional label]`

- If FAT12 is enabled, a card can be formatted with FAT12 by using the optional `[-12]` flag `format -12 [optional label]`
- If exFAT is enabled, a card can be formatted with exFAT by using the optional `[-ex]` flag `format -ex [optional label]`

Before turning our attention to FAT32 and the meaning of the information contained here, some details from the previous sections can be quickly verified. By using the command "`sect`" the content of physical sectors on the SD card can be displayed. The µtFAT formatting uses a single partition so the first sector should contain an extended boot record – which can be verified by commanding "`sect 0`"

```
>sect 0

Reading sector 0x00000000
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x02000000
 0x380b000c 0x003fb8f8 0xefc10000 0x0000003a 0x00000000 0x00000000 0x00000000 0x00000000
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0xaa550000
```

Terminal 7-4 Content of sector 0 (extended boot record)

In fact the display is of 8 rows of 16 long words (512 bytes in all) but here it has been broken down into 16 rows of 8 long words to make it fit better on the page. Apart from being aware that the values are displayed in long words in accordance to the architecture of the processor actually being run on (this is a little-endian view – a big-endian processor would display the last long word as `0x000055aa`, for example) there is no further relevance in displaying them like this for this sector's content; later however we will see that it is the natural display when interpreting FAT32 content so that is why this format was chosen generally.

If you would prefer to have the display in little-endian style when using a big-endian processors the define `UTFAT_SECT_LITTLE_ENDIAN` can be set. Alternatively the display can be set to big-endian style when using a little-endian processor or simulator using the define `UTFAT_SECT_BIG_ENDIAN`.

Here we see that there is not a great deal of content in this sector – a lot of zeros. Important is however that the last two bytes have the 0x55 0xaa pattern, indicating that the content is valid. Looking more closely at the partition information, and in particular at the value of the sector where the boot sector belonging to this partition can be found, the value `0x0000003f` can be made out (note that this is not that obvious since the locations are not on a long word boundary in the sector but the value is indeed so). The few values in the sector which are of interest to us are highlighted above.

Remembering that this partition information is just telling us where we find out boot sector we can repeat the display of the sector number 0x3f.

```
>sect 3f

Reading sector 0x0000003f
 0x4d9058eb 0x534f4453 0x00302e35 0x00200802 0x00000002 0x0000f800 0x00ff003f 0x0000003f
 0x003a9fc1 0x00000eb9 0x00000000 0x00000002 0x00060001 0x00000000 0x00000000 0x00000000
 0x9e290080 0x5553a74f 0x54414654 0x20202020 0x41462020 0x20323354 0x00002020 0x00000000
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0xaa550000
```

Terminal 7-5 Content of sector 0x3f (boot sector of the partition)

Again we can recognise that the content is marked as being valid (the 0x55 0xaa at the end of the sector) and we can interpret the content in accordance to the FAT32 sector layout. The result is that the numbers are telling us the size of the FAT32 area, where we find it and some other details – most of which have been seen in the "info" command output.

Just three value are highlighted – the values 0x08, 0x200 and 0x00000eb9. These are the values for BPB_SecPerClus, BPB_BytesPerSec[2] and BPB_FATSz32[4], whereby BPB_BytesPerSec x BPB_SecPerClus is the cluster size used by the FAT32 (8 x 512 = 4k) and BPB_FATSz32 is the FAT32 size; that is 0xeb9 (3'769) sectors, meaning that about 2 MBytes is allocated to the FAT32 area to manage the data on the disk. (In fact there are two copies of the FAT32 making total disk space used by them to be about 4 MBytes).

The FAT32 information is however not yet the real topic but you should be able to identify the bytes of information which are being used to extract various details about the formatted SD card and also being used to calculate other parameters which will later be used by the FAT32 software.

You may also have realised that the process of mounting our disk so that it is ready to be used is in fact nothing more that finding and reading this sector on the SD card, followed by interpreting a few bytes of information found there to calculate a few values as displayed by the "info" command.

But now the SD card is ready for use and it is finally time to look at what FAT32 is really all about!

7.3. Displaying the Content of a Freshly Formatted SD Card

By entering the command "dir" the following is seen.

```
>dir

Directory D:\

0 files with 0 bytes
0 directories, 1973698560 bytes free
D:\>
```

Terminal 7-6 "dir" display of empty SD card

The SD card is being displayed as disk D:\ but contains neither directories nor files. This is to be expected since the SD card has been freshly formatted and we haven't saved any data onto it yet.

Although there is no data present on the SD card there is already a small amount of information in the FAT32 area which was used by the "dir" command to determine that this is indeed the case. It is probably also obvious that the FAT32 area is what is used by the file system to manage the storage of data but it may not be clear just where this information is and what the difference is between this area and other areas on the SD card. For this reason it may be useful to review exactly what the disk area is used for since the formatting has already divided it into logical areas which are used for different functions. This is shown, based on the reference SD card, in figure 7.1.

We have already looked at the extended boot record which informs us of where the boot sector is location (sectors 0 and 0x3f). The boot sector content has then specified that the FAT32 begins at sector 0x5f and is 0xeb9 sectors in size. It has in addition informed us that there are two FAT32s (these are synchronised copies in case one were be become corrupted, although a single FAT32 would probably be adequate on an SD card since it is not susceptible to the same defects as floppy disks, for which this was originally intended for).

After the 2 FAT32 areas the cluster area begins and uses up almost all of the remaining space. This is where the data will be stored – all data management, on the other hand, is contained within the FAT32 area.

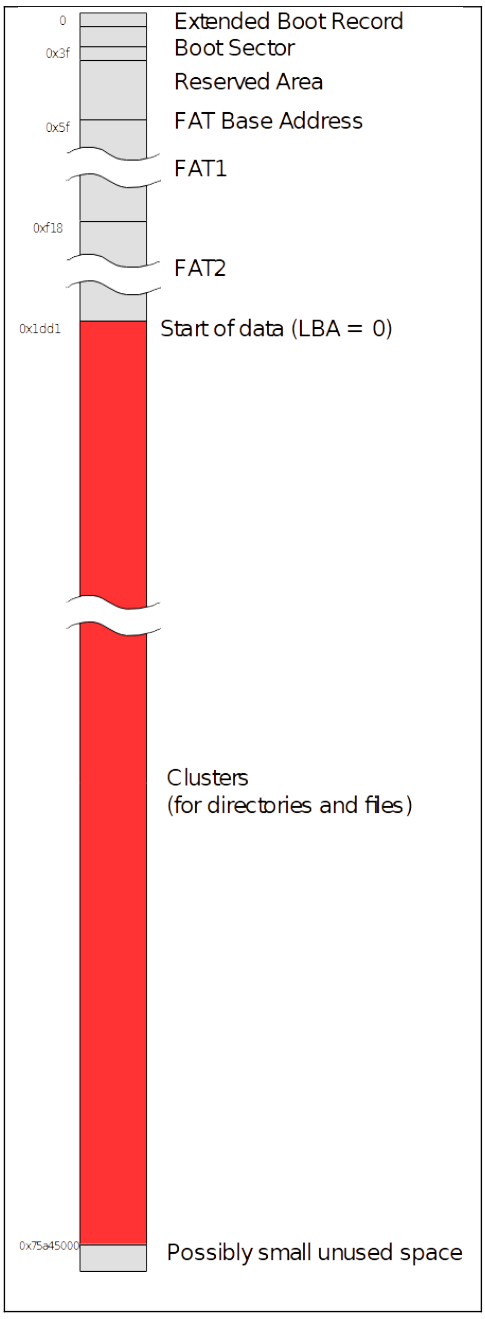


Figure 7-1 SD card/memory stick sector utilisation

There may be a few side questions at this point because there is an area at the start called the reserved area – *this is used for the boot sector but also contains (or can contain) other special information*. If you analysed the boot sector in more detail you would in fact have found that there is also a copy of the boot sector 6 sectors after the original one. There is also a special sector called the '*info*' sector at sector 0x40 which is (optionally) used by FAT32 (but not FAT12/16/exFAT) to keep extra information about which clusters are free for use – *due to the quite large size of the cluster area it can save time when having to otherwise search for specific information*. Not all of the reserved area is however necessarily used.

The other burning question may be about how the formatter decided on using a cluster size of 4k and a FAT32 size of 0xeb9 clusters. Other combinations may also be possible – for example larger cluster sizes, with less FAT32 clusters and less FAT space requirement, or smaller cluster sizes with more FAT32 clusters and greater FAT space requirement. The answer is that it is a compromise based on a simple calculation recommended by Microsoft in the *FAT32 File System Specification*. There are more details in the specification, even if a little brash (basically it states something like – *don't bother trying to understand how it works – just accept that we are right and use it* – so that is what the µtFAT disk formatter does...).

Nevertheless it is interesting to understand the relationship between FAT32 size and cluster size. A cluster is simply the smallest data chunk space that can be allocated to any object (directory structure or file data) so any object will be constructed of either 1 cluster or a multiple of such clusters. When a file is created with 1 byte content it is already occupying a cluster – in our example that is 4k of disk space; if a smaller cluster size were used it would occupy less, whereas if a larger cluster size were used it would occupy even more. Once this file's content grows to beyond a single cluster space it then starts to occupy a second cluster and can grow by simply taking as many clusters as required (until all are exhausted).

Clusters occupied by an object are not necessary contiguous. They can effectively be anywhere within the cluster space. It is the FAT32 area which contains information about where the clusters belonging to an object are physically situated. In fact it requires one long word of FAT32 space to manage one cluster (*one half-word for FAT16 and three bytes for each two for FAT12*), explaining why the FAT32 space needs to be larger when the cluster size is smaller (because it needs to be able to track more individual clusters), and vice verse. If the cluster size is chosen to be smaller it may be quite efficient for small files since they will not need to occupy as much space but larger files would need to occupy more clusters; the FAT32 would need to be larger since it has to be able to manage these additional clusters – a limit may also be reached where the FAT32 can no longer manage the total amount of clusters which can exist in the physical SD card memory. For this reason the compromise of 4k clusters and about 2 MByte FAT32, to allow effective management of almost 2GByte cluster space, turns out to be quite realistic as the following shows:

- The FAT32 size is 0xeb9 sectors in size (where a sector is 512 bytes).
- Therefore the FAT32 area is 0xeb9 x 0x200 (or 3'769 x 512) = 0x1d7200 (1'929'728) bytes in size.
- One long word (32 bits) in the FAT32 manages a single cluster (this is explained in more detail below), meaning that this FAT32 can manage 0x1d7200 / 4 = 0x75c80 (482'432) clusters.
- The cluster size is 4k so the cluster area that can be managed is 0x75c80 x 0x1000 = 0x75c80000 (1'976'041'472) bytes in size. *It will be seen that the first 2 locations in the FAT32 are not used for cluster management, actually reducing the size slightly by these 2 clusters.*
- *If we compare this with the total cluster value displayed by the "info" command there is a discrepancy and it turns out that some of the clusters don't physically exist*

(the FAT32 could manage a few more than are available). The SD card has `1'977'614'336` bytes available for use and the reserved area plus the FAT32s use up `0x1dd1` sectors (`0x3ba200` or `3'908'096` bytes), leaving `0x7545e00` (`1'973'706'240`) bytes for the cluster area. This makes `0x7404a` (exactly `481'861.875`) clusters in total for actual use. The `0.875` clusters (`3'584` bytes) are then in this case the unused bytes at the end of the SD card that couldn't be allocated to a cluster.

One additional point that is taken into account when dimensioning the FAT32 file system is to ensure that no FAT32 volume be ever configured so that a cluster `0x0ffffff7` exists. This is because this value is used to mark a *bad cluster* and so would cause a conflict (*is it a bad cluster or a cluster at that location... ?*). Since our example only needs `0x7404a` it will never be able to cause such a conflict and is thus legal.

There is now nothing standing in our way of taking a first look at the initial content of the FAT32 area, so here it is:

```
D:\>sect 5f

Reading sector 0x0000005f
 0x0ffffff8 0xffffffff 0x0fffffff 0x00000000 0x00000000 0x00000000 0x00000000 0x000...
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x000...
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x000...
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x000...
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x000...
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x000...
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x000...
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x000...
D:\>
```

Terminal 7-6 Content of first FAT32 sector – disk empty

Note that the display is in fact 16 x 8 long words in size (512 byte sector size) which will turn out to be useful as soon as cluster chains need to be followed.

All long words with the value `0x00000000` represent free cluster space which can be used when allocating clusters.

The first two long word values are standard entries which are described in the Microsoft *FAT32 File System Specification* but are otherwise of no significance to us. They are never used but need to be there. The third long word `0x0fffffff` is a FAT32 entry which is used by the *root directory*. Although the disk is empty it always has a root directory – our root directory is `D:\` Note that the value `0x0fffffff` means that the cluster is (presently) a single cluster – *later we will see how the FAT32 area manages cluster chains*.

This is always the first cluster. Since it is however not the first occupied FAT32 entry it is referred to as cluster 2. Cluster 2's cluster location is the first cluster in the physical cluster area, whose address is referred to as the LBA (Logical Base Address). *The fact that cluster entry 2 is in fact physical cluster 0 can be a little confusing and also explains why there is often a conversion of -2 or +2 when translating between cluster entries and cluster locations... The µtFAT module avoids the conversion difficulties by maintaining two base address values; the 'logical' base address and the 'virtual' base address. The 'logical' base address is used when working with sectors within a cluster, relative to where the cluster area physically starts. The 'virtual' base address is used when working with clusters since it automatically references it to two clusters before the physical cluster start and so avoids any additional need to compensate for the unused cluster entries.*

Since we now know that the root directory already has its own cluster entry we can take a look at this too. It is situated at the very start of the cluster area (LBA) which starts at sector `0x1dd1` (see "`info`" command). *Don't forget that it does in fact occupy at least one cluster space so its content is not sector `0x1dd1` alone but also the following sectors up to and including `0x1dd9` (with `4k` cluster size each cluster is made up of `8` sectors of each `512` bytes in size).*

```
D:\>sect 1dd1

Reading sector 0x00001dd1
 0x41465455 0x00000054 0x08000000 0x92370000 0x40634063 0x92370000 0x00004063 0x000...
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x000...
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x000...
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x000...
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x000...
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x000...
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x000...
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x000...
D:\>
```

Terminal 7-7 Content of empty root directory

This sector's content is in the cluster area so has nothing to do with FAT32 management. Its content is described by a directory entry, consisting of `32` bytes. A single directory could thus contain `128` entries (generally files or further directories) before its first `4k` cluster is fully occupied and a second cluster is required. The directory entry content is shown below:

```
typedef struct stDIR_ENTRY_STRUCTURE_FAT32
{
    unsigned char DIR_Name[11];       // directory short name. If the first byte is 0xe5 the
                                         directory entry is free. If it is 0x00 this and all
                                         following are free. If it is 0x05 it means that the
                                         actual file name begins with 0xe5 (makes Japanese
                                         character set possible). May not start with ' ' or
                                         lower (apart from special case for 0x05) and lower
                                         case characters are not allowed. The following
                                         characters are not allowed: "0x22, 0x2A, 0x2B, 0x2C,
                                         0x2E, 0x2F, 0x3A, 0x3B, 0x3C, 0x3D, 0x3E, 0x3F, 0x5B,
                                         0x5C, 0x5D, and 0x7C
    unsigned char DIR_Attr;           // file attributes
    unsigned char DIR_NTRes;          // reserved for Windows NT - should be 0
    unsigned char DIR_CrtTimeTenth;   // millisecond stamp at file creation time. Actually
                                         contains a count of tenths of a second 0..199
    unsigned char DIR_CrtTime[2];     // time file was created
    unsigned char DIR_CrtDate[2];     // data file was created
    unsigned char DIR_LstAccDate[2];  // last access date (read or write), set to same as
                                         DIR_WrtDate on write
    unsigned char DIR_FstClusHI[2];   // high word of this entry's first cluster number
                                         (always 0 for a FAT12 or FAT16 volume)
    unsigned char DIR_WrtTime[2];     // time of last write, whereby a file creation is
                                         considered as a write
    unsigned char DIR_WrtDate[2];     // date of last write, whereby a file creation is
                                         considered as a write
    unsigned char DIR_FstClusLO[2];   // low word of this entry's first cluster number
    unsigned char DIR_FileSize[4];    // file's size in bytes
} DIR_ENTRY_STRUCTURE_FAT32;
```

Code 7-1 Directory entry struct

Analysing the single directory entry in the root directory reveals the volume name "`UTFAT`", with attribute "volume ID" (0x80). There are no further entries because the following directory entry starts with a `0x00`. The creation data is also set, whereby this will be the local time if the define

```
#define SUPPORT_FILE_TIME_STAMP
```

is set and there is a corresponding source of time and date (usually used together with real-time-clock support) or else will be a fixed time and date. *The time and data details are shown with black background*.

7.4. Creating a New Directory

These descriptions assume that directories and files are in 8:3 format. That is, they correspond to short file names like "FILETEST.TXT". The effect of long file names is discussed later on in the document.

A new directory can be created by commanding "`mkdir dir1`". This will create the directory `dir1` in the root directory which is then listed when the "`dir`" command is executed.

```
D:\>mkdir dir1

D:\>dir

Directory D:\

---- 26.03.2024  15:40 <DIR>            dir1
0 files with 0 bytes
1 directories, 1973698560 bytes free
D:\>
```

Terminal 7-8 Creating and listing a new directory

The new directory entry can now also be seen in the root directory's cluster:

"UTFAT" – Volume ID "DIR1" – Directory "DIR1" starts in cluster 3

```
D:\>sect 1dd1

Reading sector 0x00001dd1
 0x41465455 0x00000054 0x08000000 0x92370000 0x40634063 0x92370000 0x00004063 0x000...
 0x31524944 0x20202020 0x10202020 0x92f60018 0x40634063 0x9f600000 0x00034063 0x00000000
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x000...
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x000...
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x000...
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x000...
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x000...
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x000...
D:\>
```

Terminal 7-9 Content of the root directory with one directory

The attribute 0x10 indicates that the entry is a 'directory' and the cluster location of this directory set to `0x00000003`.

µtFAT also adds a time and date stamp when directories and files are created and when they are written as described in the previous section (the part with black background).

The value 0x18 (`DIR_NTRes`) is always set although it is generally described as being 0; this is consistent with the behaviour of the FAT file system used by Windows but the reason for it is not known.

The new directory has also been allocated its own cluster, which can also be seen in the FAT32:

Specifies single cluster used by *root directory*

Specifies single cluster used by directory *dir1*

```
D:\>sect 5f

Reading sector 0x0000005f
 0x0ffffff8 0xffffffff 0x0fffffff 0x0fffffff 0x00000000 0x00000000 0x00000000 0x000...
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x000...
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x000...
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x000...
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x000...
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x000...
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x000...
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x000...
D:\>
```

Terminal 7-10 Content of first FAT32 sector – one directory in the root directory

The newly created cluster for the directory "`dir1`" is also not completely empty as shown below. Note that the cluster `3` is one cluster after the root directory's, meaning that it starts at `0x1dd1` + `8` = `0x1dd9` (remembering that each `4k` cluster is made up of `8` sectors).

"." – Directory

".." - Directory

```
D:\>sect 1dd9

Reading sector 0x00001dd9
 0x2020202e 0x20202020 0x10202020 0xa1ca0000 0x40634063 0xa1ca0000 0x00034063 0x00000000
 0x20202e2e 0x20202020 0x10202020 0xa1ca0000 0x40634063 0xa1ca0000 0x00024063 0x00000000
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x000...
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x000...
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x000...
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x000...
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x000...
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x000...
D:\>
```

Terminal 7-11 Content of the empty directory "dir1"

Analysing the directory entries reveals that there are in fact two directories called '.' and '..' automatically created and the rest of the directory cluster is set to 0. These are in fact also displayed when the SD card is used in a PC to indicate the present directory and the path upward to the next higher directory. The µtFAT module generates these for compatibility but doesn't actually use them.

Note that the clusters entered for the two entries are the same cluster for "." and the cluster of the upper level for ".." – the directory "." is therefore the own directory and the directory ".." is one level up!

Here we have seen that the cluster 3 was allocated for use by the new directory. This was simply the next available free one but, theoretically, it would be possible to allocate "any" free one.

7.5. Creating a New File

These descriptions assume that directories and files are in 8:3 format. That is, they correspond to short file names like "FILETEST.TXT". The effect of long file names is discussed later on in the document.

Although it is possible to create an empty file by using the "`file`" command, and also adding some content to it (256 bytes each execution) using the "`write`" command we will be a little more adventurous for the next steps. This assumes that the board used for tests also has an Ethernet connection since the µTasker FTP and HTTP servers will now come into play.

1) Connect to the board from a DOS window via FTP. Check that the same single directory is displayed when the "`dir`" command is executed.
2) Now change directory with "`cd dir1`". You are now in the new directory, which is presently empty.
3) Transfer an existing HTM file (it is assumed that this is called "`test.htm`") which is larger than `4k` in size (this will ensure that it uses more than one cluster to be saved in) and save it as "`index.htm`" – using the command "`put test.htm index.htm`".
4) Browse to the IP address of the board to view this file.

Note that, if you have no `.htm` file suitable you can also transfer a file such as a PDF document and access it by browsing directly to it with http://192.168.0.3/test.pdf, for example.

It is assumed that the FTP and HTTP tests went well – details of working with FTP and HTTP are described in later sections. However the point of the test was in fact to add a file to the sub-directory and see what the FAT32 area looks like now.

```
D:\>sect 5f

Reading sector 0x0000005f
 0x0ffffff8 0xffffffff 0x0fffffff 0x0fffffff 0x00000005 0x0fffffff 0x00000000 0x000...
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x000...
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x000...
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x000...
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x000...
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x000...
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x000...
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x000...
D:\>
```

Terminal 7-12 Content of first FAT32 sector – including one file occupying multiple clusters

This shows the result of saving a file of `5'199` bytes (`0x144f`) in length. It was originally created in cluster 4 (it started out with `0x0fffffff`) but then overflowed and required a new cluster. Since the next cluster was free it then started occupying cluster 5, which has the value `0x0fffffff` since it is indeed the last in the new cluster chain.

The cluster 4 entry has however been modified in the process to indicate where the following cluster can be found – in this case simply in the next cluster space, cluster number 5, but it could be anywhere in the cluster area.

When the file was served by the HTTP server the file needed to be first found – this was achieved by searching the directory for the file entry so that the length of the file and its cluster location became known. Below is the directory's content again after the new file "`index.htm`" was added:

"INDEX.HTM" – File File content starts in cluster 4

```
D:\>sect 1dd9

Reading sector 0x00001dd9
 0x2020202e 0x20202020 0x10202020 0xa1ca0000 0x40634063 0xa1ca0000 0x00034063 0x00000000
 0x20202e2e 0x20202020 0x10202020 0xa1ca0000 0x40634063 0xa1ca0000 0x00024063 0x00000000
 0x45444e49 0x20202058 0x204d5448 0xa4d50018 0x40634063 0xa4d50000 0x00044063 0x0000144f
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x000...
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x000...
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x000...
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x000...
 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x000...
D:\>
```

Terminal 7-13 Content of the empty directory "dir1" with new file "index.htm"

The file is recognised by its file attribute 0x20, has a length of `0x144f` and its start is in cluster `4`.

Therefore the file was found by the HTTP server and the content could be read. During the read process the first `4k` file content was taken from cluster `4`, which needed no FAT32 intervention. But, as soon as the complete content of the first cluster had been read it was necessary to check where the rest could be found – *the fact that it is simply in the following cluster is an assumption that cannot be made since it could in fact be anywhere in the cluster area!*

The way that this took place was to consult the FAT32 table by seeing whether this is the final cluster (a value of `0x0fffffff` would be expected if that was the end of the cluster chain belonging to the file) or whether there are following clusters. Since this present cluster is `4` (counting from `0`) its corresponding FAT32 entry can be found by reading the first FAT32 sector and then using the 4th cluster entry (see Terminal 7.12) to see which cluster to use next. In this case it does simply use the following cluster number `5`.

Having the sector display as a 16 x 8 field makes following FAT32 cluster chains quite simple since the present cluster can be read directly – eg. if the present cluster is `0x23`, its corresponding entry is two lines down and 4 entries to the right [*in the case above (0x04) it was 0 lines down and the 4th entry to the right*]. If this is `0x0fffffff` it is the final cluster in the chain, else the next one is directly displayed.

Generally the calculation for a cluster in a FAT32 file system is as follows:

1) The FAT32 sector in which the cluster entry is to be found is (`cluster/128`). This gives the actual physical sector as `FAT_start` + (`cluster/128`), where `128` comes from the number of long words in a sector (`512/4`)
2) The entry in this sector is (`cluster & 0x7f`), which is a value between `0..127`.

FAT32 allows a single file of up to almost 4 GByte in size to be saved and retrieved. A cluster chain can be followed by using the technique above to move from the start of the file - one cluster at a time - until the end of the file is reached.

Note that FAT32 doesn't allow for efficiently moving backwards through a cluster chain!

Clusters on SD cards (and memory sticks) don't take longer to access when located closer or further apart. Floppy disks however do since the locations require physical movement to take place which takes longer as the distance increases. Defragmentation is a technique used to improve the access speed on floppy disks and hard discs, whereby the clusters in single files can be moved to occupy a contiguous section of memory. *Defragmentation of SD cards and memory sticks has, in comparison, no benefits.*

When FAT16 is used the basic operation is very similar and the main difference is that the FAT table doesn't use 32 bit entries but rather uses 16 bit entries. This means that the longest cluster chain is limited to about 64k rather than 255M using FAT32. FAT16 is usually used only when the storage space is relatively small due to the cluster count limitation – its FAT table is smaller than a corresponding FAT32 table since each cluster entry is half the size but operations based on the FAT16 table tend to be more complicated than FAT32 operations because the storage elements are not so natural as used by todays 32 bit processors. FAT12 uses 12 bit entries and so it is even more complicated to calculate and maintain the entries and FAT12 is rarely used today since it is only suitable for very small file system sizes.

It is to be noted that the FAT format used can be read from the boot sector details, which differ slightly in each case.

Finally, FAT16 and FAT12 have an awkward restriction concerning the root directory which has a fixed size and cannot grow as it can with FAT32. This fixed size means that a certain number of files (fixed 16k size is common and used by the µtFAT implementation) can fit in the root directory. As has already been seen, a single entry (directory, file) occupies 64 bytes of space and so the limit is 256 such entries. As will however been seen later, directories and files using long file names may occupy multiple entries to save the complete name information, meaning that only a reduced number of elements using long names can fit. The limitation can be easily seen in operation on a FAT16/FAT12 formatted SD card when an attempt is made to copy a large number of individual files to its top directory; the attempt results in an error message from Windows and the user needs to create a sub-directory to achieve the storage.

Due to the fact that FAT16/FAT12 have additional complications but only negligible memory advantages in most modern systems it is recommended to generally use FAT32 and exFAT for very large SD cards or memory sticks.

8. FTP Server and µtFAT

The FTP server can work with the SD card/memory stick if the define FTP_UTFAT is active. If a formatted device is inserted accesses will be to this. The root directory used by the FTP server is defined by FTP_ROOT "/"

In this case it has root access and so can work in all directories and sub-directories on the disk. More restricted access can also be given if required (the FTP root being set to a sub-directory).

Since µtFAT V2.1 supports reading and displaying long file names (with define UTFAT_LFN_READ active) it can display such folders and files which were copied to the disk from a PC supporting LFN. Furthermore if UTFAT_LFN_WRITE is enabled LFNs can be written or renamed. UTFAT_LFN_DELETE enables also LFN deletes.

The FTP interface supports moving between directories, creating directories, writing and reading files (writes truncate files, meaning that existing files will first be deleted), deleting files and empty directories and renaming files and directories.

Accesses are always relative to the present directory position.

If the SD card/memory stick is removed the FTP server will fall back to work with the µFileSystem.

9. HTTP Server and µtFAT

The HTTP server can work with the SD card/memory stick if the define HTTP_UTFAT is active. If a formatted device is inserted accesses will be to this. The root directory used by the HTTP server is defined by HTP_ROOT "dir1"

In this case it has access to all directories and sub-directories in this directory but not higher.

Since µtFAT V2.1 supports reading long file names (with define UTFAT_LFN_READ active) it can serve linked files from directories with long file names which were copied to the disk from a PC supporting LFN. The HTTP root can also be a long file name path in this case.

Accesses to files are always relative to the HTTP root.

If the SD card/memory stick is removed the HTTP server will fall back to work with the µFileSystem.

The default file served when no file is defined (default file) is defined by DEFAULT_HTTP_FILE "index.htm". This can also be a long file name if the option UTFAT_LFN_READ is active

The HTTP release at the time of writing doesn't support posting data to the SD card/memory stick – this is however not a restriction of the µtFAT module but missing native support in the HTTP server. The application can however intercept such posted data if required to save it to an SD card/a memory stick.

A powerful feature of the HTTP server and long file name support is that existing web server content can be copied from a PC to an SD card/memory stick, which can then be served by the embedded board. The only requirement is that the HTTP root directory corresponds to that configured by the project and that the default file exists with the correct name. All standard web content can then be server / browsed, including web pages, images,

documents etc. Due to the large size of the SD card/memory stick memory very large web server contents can be used even together with small processors.

Since the µTasker doesn't support server side technologies like PHP, such file types cannot be handled. However the µTasker server's dynamic HTTP methods can be applied to such HTTP content if required.

10. Working with the µtFAT User Interface

The µtFAT user interface shows how moving around directory contents can be performed in a simple manner similar to the well known DOS interface. Terminal 10.1 shows the contents of a sub-directory which is being displayed from a higher level directory – notice that the user's position is not in the directory being displayed but one level higher, thus the command "`dir webpages/webpagesSAM7x`" includes the full relative path to the directory.

```
>dir webpages/webpagesSAM7x

Directory webpages/webpagesSAM7x

---A 29.07.2009  13:39             2452 0Menu.HTM
---A 25.07.2009  23:36             2600 4Lan.htm
---A 06.09.2006  22:48             2007 7Logo.jpg
---A 25.07.2009  23:37             2977 9I_O.htm
---A 25.07.2009  23:27             1114 CLCD.htm
---A 25.07.2008  00:47               40 Copy_all.bat
---A 04.05.2008  19:04               44 delete_all.bat
---A 25.07.2009  23:38             1620 EStats.htm
---A 26.07.2009  00:25              195 ftp.txt
---A 23.04.2008  16:19               33 ftp_del.txt
---A 25.07.2009  23:39             2338 Hserial.htm
---A 25.07.2009  23:42             2019 Kadmin.htm
---A 25.07.2009  23:48             1578 Mhelp.htm
---A 07.10.2006  13:50             2498 OLogo.gif
---- 29.07.2009  13:43 <DIR>            AlternativePages
---- 20.06.2008  13:50 <DIR>            Alternative_Logo
---- 29.07.2009  13:39 <DIR>            FileSystem
14 files with 21515 bytes
3 directories, 3173023744 bytes free
>
```

Terminal 10-1 Content of an SD card containing directories and files using LFN

As shown in Terminal 10-2 path names can use '/' or '\', and file names are not case sensitive

```
>print webpages/webpagesSAM7X\Mhelp.htm

<html><head><meta http-equiv="content-type" content="text/html; charset=UTF-8"><title>&micr..
<table cellspacing=0 cellpadding=0><tr><td width=10%> </td><td align=left><br><br>
Thank you for using the <b>&micro;Tasker</b> for the SAM7X.<br><br>
I hope that you have fun and can save time using it when developing your own applications....
To get latest details about the <b>&micro;Tasker</b> developments for this and further pla...
<b>M.J.Butcher Consulting</b><br>
Birchstrasse 20f,<br>
CH-5406 Baden-Ruetihof,<br>
Switzerland<br><br>
+41 (0)56 535 15 70<br><a
href="mailto:info_uT@uTasker.com">info_uT@uTasker.com</a><br><br><br>
The <B>&micro;Tasker</B> is available free of charge, including email support, for educati...
</font></td></tr></table><br><br>
<a href="0Menu.htm"> Go back to menu page</a>
</body></html>
```

Terminal 10-2 Printout of the content of a file in a referenced directory

Directories can be changed by using the "cd" command, for example "cd webpages". Moving upwards is possible with "cd ..", or "cd ../..", etc.. Referencing other directories not under the present directory location can be achieved by "cd ../../webpages", for example.

The complete list of basic commands are (commands are case-sensitive):

info	utFAT card info	Display SD card/Memory stick and FAT information
dir	[path] show directory content	Display present directory, eg. "dir dir1/webpages"
cd	[path] change dir. (.. for up)	Move the directory location, eg. "cd dir/webpages"
comp	compare [file1] with [file2]	Compare the content of two files
file	[path] new empty file	Create an empty file. If a file exists with the same name it will be truncated, meaning that its length will be set to 0 and its original content effectively destroyed.
print	[path] print file content	Print the content of an existing file to the debug output (non-displayable characters are output as '.')
sect	[hex number] display sector	Display the content of a the defined 512 bytes sector on the SD card/memory stick, eg. "sect 5f"

Dependent on UTFAT_WRITE

file	[path] new empty file	Create an empty file. If a file exists with the same name it will be truncated, meaning that its length will be set to 0 and its original content effectively destroyed.
write	[path] test write to file	Write test data to the file – the file's length is increase each time by 256 bytes with a value 0x55 the first time used, 0x56 the second, etc.
mkdir	new empty dir	Create a new directory, eg. "mkdir dir2". If the directory already exists it will not be modified.
rename	[from] [to] rename	Rename an existing file or directory, eg. "rename dir1 dir2"
trunc	truncate to [length] [path]	Truncate an existing file, eg. "trunc file1.txt 1024" The file will be reduced in length to the new length defined. The number of clusters that

		were freed will be displayed.
copy	[file1] to [file2]	Copy the content of the first file to the second file
hide	[path] file/dir to hide	Set a file's property to a hidden file
unhide	[path] file/dir to un-hide	Set a file's property to visible file
prot	[path] file/dir to write-protect	Set a file's property to a write-protected file
unprot	[path] file/dir to un-protect	Set a file's property to remove write-protection
del	[path] delete file or dir.	Delete a file or directory. Directories can only be deleted when they are empty.
format	[-16/12/ex] [label] format (unformatted) disk	Format an unformatted SD card with optional volume ID, eg. "format UTFAT_DISK" [-16] only possible when FAT16 is supported [-12] only possible when FAT12 is supported [-ex] only possible when exFAT is supported When nothing is defined FAT32 format is used
fformat	[-16/12/ex] [label] FULL format (unformatted) disk	Same as "format" but deletes all cluster content – only available with option UTFAT_FULL_FORMATTING
re-format	[-16/12/ex] [label] reformat disk!!!!! **WARNING – the existing FAT32 table will be destroyed!!**	Reformat a formatted disk, with optional volume ID, eg. "format UTFAT_DISK" [-16] only possible when FAT16 is supported [-12] only possible when FAT12 is supported [-ex] only possible when exFAT is supported When nothing is defined FAT32 format is used
re-fformat	[-16/12/ex] [label] FULL reformat disk!!!!! **WARNING – the existing FAT32 table and all disk content will be destroyed!!**	Same as "re-format" but deletes all cluster content – only available with option UTFAT_FULL_FORMATTING

Dependent on TEST_SDCARD_SECTOR_WRITE in debug.c.

sectw	[hex number] [pattern]	Write a pattern (bytes starting with the value pattern and incrementing for each byte written) to a specified section. This command is only available for low-level test purposes when the define is enabled

Dependent on UTFAT_UNDELETE || UTFAT_EXPERT_FUNCTIONS

dird	[path]	show deleted directory content (rather than valid content)

Dependent on UTFAT_EXPERT_FUNCTIONS

dirc	[path] show corrupted directory content	show result of analysis of any file corruption found in a directory
dirh	[path] show hidden directory content	show hidden directory content (rather than visible content)
infof	[path] show file info	shows detailed information about the file or directory's object and storage, including SFN and LFN content and physical location
infod	[path] show deleted info	shows detailed information about the deleted file or directory's object and storage, including SFN and LFN content and physical location

Dependent on UTFAT_SAFE_DELETE

dels	[path] safe delete file or dir.	Deletes entire file details and content so that they can neither be seen by "dird" or recovered by "undel"

Dependent on UTFAT_UNDELETE

undel	undelete [name]	Undelete a deleted file (as displayed by "dird") and give it the name UNDELETE.TXT, which can subsequently be renamed if required Example "undel ~test1.txt"

11. µtFAT Application User Interface

The following commands are used by the disk interface (see debug.c), which serves as additional reference to their use.

11.1. utAllocateDirectory()

```
extern UTDIRECTORY *utAllocateDirectory(unsigned char ucDisk, unsigned short usPathLength);
```

This function allocates a directory from the directory pool (size of pool defined by UT_DIRECTORIES_AVAILABLE in config.h). For the SD card the disk should always be DISK_D. For a memory stick DISK_E is used.

The pointer returned is to the directory object allocated for use.

The value of usPathLength is the length of a path string that is to be used with the directory. This size can be 0 if none is required otherwise the string space is also allocated on heap for use by the object (ptr_utDirectory->ptrDirectoryPath).

It is recommended to use a directory pointer as only interface to the disk (SD card or memory stick).

```
static UTDIRECTORY *ptr_utDirectory = 0;           // pointer to a directory object
```

Often one single directory pointer is adequate for a particular interface (for example the HTTP server uses just one to control all of its possible HTTP sessions) – there is no limit to the quantity of files that can be opened using the single directory pointer.

The first action is to register the directory, which is performed only once:

```
if (ptr_utDirectory == 0) { // if not yet allocated
    ptr_utDirectory = utAllocateDirectory(DISK_D, UT_PATH_LENGTH);
                          // allocate a directory for use by this module
                          // associated with D: and reserve its path name string length
}
```

DISK_D is always used for the SD card and a path length is chosen that will be adequate to hold the complete path string to the directories used*. This string length can also be 0 if all referencing is from the present directory (eg. HTTP doesn't have a path string but instead all is referenced to its own root).

* When using a string it takes up space which is created within utAllocateDirectory() but allows moving up and down a directory path (assuming starting at the root d:/):

```
cd dir1 (new position = d:/dir1/)
cd dir2 (new position = d:/dir1/dir2)
cd dir3 (new position = d:/dir1/dir2/dir3)
cd .. (new position = d:/dir1/dir2 [the .. moves up one level])
cd ../dir5 dir2 (new position = d:/dir1/dir5)
```

The length needs to be adequate for the deepest directory (d:/dir1/dir2/dir3/dir4/dir5/... etc.). Note that the DOS-like menu interface uses a path string to accomplish this.

FTP also uses a path string. It can reference directives below itself "dir dir2/dir3/dir4" and it can also move down "cd dir2/dir3" as well as move back up the path or reference upwards like "cd ../dir5".
Therefore the string use is user definable. HTTP can work without needing a directory path string (saves space) but the DOS like interface uses one since it makes it easier for the user to move around with.

11.2. utOpenDirectory()

```
extern int  utOpenDirectory(const CHAR *ptrDirPath, UTDIRECTORY *ptrDirObject);
```

This function sets the root directory location for the directory object. The path, ptrDirPath is a path reference relative to the root directory of the disk. It can be 0 if the directory object's root position is to be equal to the disk's root directory.

The routine returns UTFAT_SUCCESS if the directory location could be set.

Errors can be:

UTFAT_DISK_NOT_READY - disk not ready for use - not formatted or not mounted
UTFAT_PATH_NOT_FOUND - the referenced directory path could not be found
UTFAT_DISK_READ_ERROR - error occurred while trying to read a sector from the disk
UTFAT_PATH_IS_FILE - the referenced object is a file and not a directory

The directory pointer is validated by setting its root directory. This directory can be the root of the disk or any existing directory or sub-directory on the disk.

```
if (utOpenDirectory(0, ptr_utDirectory) != UTFAT_SUCCESS) {   // open the root directory
    fnDebugMsg("No SD-Card ready\r\n");
}
```

The root directory can be specified by using 0, '\' or '/'. Directories or sub-directories can be specified by entering the full path to its location:

```
if (utOpenDirectory("/HTTP_DIR", ptr_utDirectory) != UTFAT_SUCCESS) {// open the directory
    fnDebugMsg("Directory not found\r\n");
}
```

To check to see whether the directory pointer is valid for operation the check

```
if ((ptr_utDirectory->usDirectoryFlags & UTDIR_VALID) != 0) {}
```

can be used. The opened directory is now the highest location that can be accessed together with the directory object and is also the start of the optional path string.

The directory flags are:

UTDIR_ALLOCATED	set if the directory object is allocated to an application
UTDIR_VALID	set if the directory object is valid and can be used
UTDIR_REFERENCED	set if the present access is referenced to the present directory path location
UTDIR_SET_START	if the file is not found the directory should be set to the lower directory in the path so that new files can be added there
UTDIR_DIR_AS_FILE	if set handles directories and files equivalently - used when renaming and deleting
UTDIR_TEST_REL_PATH	if set, test a path relative to the present directory path location but don't move to it
UTDIR_TEST_FULL_PATH	signifies temporary reference from the root directory
UTDIR_TEST_FULL_PATH_TEMP	signifies temporary reference from the present directory path reference
UTDIR_ALLOW_MODIFY_PATH	if set, allow a directory search to modify the directory path string if it exists

11.3. utChangeDirectory()

```
extern int  utChangeDirectory(const CHAR *ptrDirPath, UTDIRECTORY *ptrDirObject);
```

This function moves the present directory location (originally set with utOpenDirectory()) to a new one.

If the command is successful it returns UTFAT_SUCCESS.
Errors can be:
UTFAT_PATH_NOT_FOUND – new path is invalid
UTFAT_DISK_NOT_READY – disk not ready (eg. it has been removed)
UTFAT_DISK_READ_ERROR - error occurred while trying to read a sector from the disk

The new directory location is referenced to the present directory and the file object's ptr_utDirObject must be set to the directory pointer.

```
if (utChangeDirectory("../../dir1/dir2", ptr_utDirectory) != UTFAT_SUCCESS) {
                                                          // change the directory location
    fnDebugMsg("Invalid path\r\n");
}
```

This example shows a movement from a sub-directory location up two levels and then into dir1/dir2.

11.4. utOpenFile()

```
extern int  utOpenFile(const CHAR *ptrFilePath,
                       UTFILE *ptr_utFile,
                       UTDIRECTORY *ptr_utDirectory,
                       unsigned long ulAccessMode);
```

This function opens a file referenced by ptrFilePath and belonging to the directory ptr_utDirectory with the access mode as defined by usAccessMode.
The access modes are one or more of these:
UTFAT_OPEN_FOR_READ - file to be opened for reading
UTFAT_OPEN_FOR_WRITE - file to be opened for writing to
UTFAT_OPEN_FOR_DELETE - file to be opened so that it can be deleted
UTFAT_PROTECTED - the file is to be opened and protected - no access by other users
UTFAT_MANAGED_MODE - open the file in managed mode so that any changes to it by other users are automatically updated
[*the following are flags that can be used to control the open but not saved as file object mode*]
UTFAT_OPEN_FOR_RENAME - file to be opened so that it can be renamed
UTFAT_TRUNCATE - if the file already exists truncate it so that its length is zero
UTFAT_CREATE - if the file doesn't exist create it
UTFAT_APPEND – opens a file and automatically sets its file pointer to the end of any existing file ready for subsequent append writes
UTFAT_DISPLAY_INFO – only available with option UTFAT_EXPERT_FUNCTIONS. Used to request details of file to be printed to the debug output
UTFAT_COMMIT_FILE_ON_CLOSE – the file's details (size, time, etc.) is only updated to disk when the file is closed. This avoids writes to the disk on each data write (can greatly improve write speeds) with a risk that the file's size is not correct if there is a system reset before the close.
UTFAT_WITH_DATA_CACHE - only available with option UTFAT_FILE_CACHE_POOL. The file is allocated a sector buffer (when available) and modified data is only committed to the disk when the sector is changed. Reads from the sector (also from other users) are taken from the sector buffer (data cache) when its content is up to date. This can improve read speed if the reads access cached data.

If the open command is successful it returns UTFAT_PATH_IS_FILE. Note that this a positive value but explicitly identifies a file open with this value. The file object pointed to by ptr_utFile is filled out with the file information for use by the application.

Errors can be:
UTFAT_DISK_NOT_READY – disk not ready; for example, the card was removed
UTFAT_SEARCH_INVALID - a file search was invalid since the file object is not associated with a directory object
UTFAT_PATH_NOT_FOUND – invalid path was entered
UTFAT_DISK_WRITE_PROTECTED – open for write, delete or rename failed since the file is read only
UTFAT_FILE_NOT_FOUND - the referenced file was could not be found
UTFAT_FILE_LOCKED - the file could not be opened since it is locked for exclusive use by another user
MANAGED_FILE_NO_FILE_HANDLE – no space is available for a managed file
UTFAT_DISK_READ_ERROR - error occurred while trying to read a sector from the disk

A file can be opened for read and/or write access. It is always opened referenced to the present directory. [*Note that the directory pointer does not have to be entered to the `UTFILE` struct from utFAT2.0 since the it is now passed to `utOpenFile()`*]

```
UTFILE utFile;                                          // temporary file object
if (utOpenFile(ptrInput, &utFile, ptr_utDirectory,
               (UTFAT_OPEN_FOR_READ | UTFAT_OPEN_FOR_WRITE |
                UTFAT_CREATE)) != UTFAT_PATH_IS_FILE) {
                      // open a file for reading and writing and create if not existing
    fnDebugMsg("Create file failed\r\n");
}
else {
    fnDebugMsg("File length = ");
    fnDebugDec(utFile.ulFileSize, 0);
    fnDebugMsg("\r\n");
}
```

When opening a file it can be created if not already existing and can be truncated (content deleted if existing).

A file opened in managed mode should also have the `ownerTask` element of the file object set – this is used to identify the task “owning” the opened file (there can however be multiple owner tasks)

```
utFile.ownerTask = OWN_TASK;
```

A file object in managed file mode will be automatically updated if any user modifies the opened file. The user modifying the file doesn’t have to open it in managed mode for this to happen. For example, if another user writes to the file, the file length will be automatically adjusted accordingly in the file object. If another user should reduce the length of a file its length in the file object will also be adjusted and the file pointer may also be corrected so that it is in a valid range.

Generally the file object is used for subsequent file operations, like writes and reads. The object contains two values which the user often accesses to gain information about the file:

```
utFile.ulFileSize;                  - file's total length
utFile.ulFilePosition;              - present linear file position (pointer)
```

11.5. utTruncate()

```
extern int utTruncateFile(UTFILE *ptr_utFile);
```

This function allows an existing file, reference by `ptr_utFile`, to be truncated to a new, smaller length. Its typical use is to move the file pointer to a specific location in the existing file and then limit the file length to that location, whereby all additional content (clusters) that were used by the original file size are freed.

The routine returns `UTFAT_SUCCESS` if the truncation was successful. It also returns success if the file pointer was already located at the end of the existing file.

Errors can be:
`UTFAT_FILE_NOT_WRITEABLE` - file not writeable
`UTFAT_DISK_READ_ERROR` - error occurred while trying to read a sector from the disk

The following example shows an existing file being opened and then the file pointer being set to a fixed location (assumed to be at a position shorter than the original file size). After the truncation the file will be reduced to this size and so further writes to the file effectively add new data from this location.

```
utOpenFile("test_file.txt", &utFile, ptr_utDirectory,
UTFAT_OPEN_FOR_WRITE);
utSeek(&utFile, 100, UTFAT_SEEK_SET);
utTruncateFile(&utFile);
```

11.6. utSeek()

```
extern int utSeek(UTFILE *ptr_utFile, unsigned long ulPosition, int iSeekType);
```

This function controls the position of the file pointer within the file referenced by `ptr_utFile`.
The following `iSeekType` values are valid:
`UTFAT_SEEK_SET` - set to the position relative to the start of the file
`UTFAT_SEEK_CUR` - set to the position relative to the present position (can be positive or negative)
`UTFAT_SEEK_END` - set to the position relative to the end of the file

The routine returns `UTFAT_SUCCESS` if the new pointer location could be set.

Errors can be:
`UTFAT_DISK_READ_ERROR` - error occurred while trying to read a sector from the disk
`UTFAT_DIRECTORY_AREA_EXHAUSTED` - the end of the FAT space was reached and no valid clusters found

When a file is opened (for reading or for writing) its internal file pointer is generally set to the start of the file. If additional data is to be written to a file, without overwriting existing data at the start of the file, the file pointer can be positioned to the end of the file by using the command

```
utSeek(&utFile, 0, UTFAT_SEEK_END);
```

The use of `utSeek()` is equivalent to the well known `lseek()` library function.

Note that opening a file with the flag `UTFAT_APPEND` automatically causes a seek to the end of the file to take place.

11.7. utWriteFile()

```
extern int utWriteFile(UTFILE *ptr_utFile,
                       unsigned char *ptrBuffer,
                       unsigned short usLength);
```

This function allows content (binary) to be written to the file referenced by `ptr_utFile`. The amount of data `usLength` from `ptrBuffer` will be written to the file beginning from the present file pointer position. The length of the file will be automatically increased if the data write is beyond the present file end.

The routine returns `UTFAT_SUCCESS` if the data was successfully written.

Errors can be:
`UTFAT_DISK_READ_ERROR` - error occurred while trying to read a sector from the disk
`UTFAT_FILE_NOT_WRITEABLE` - the file cannot be written because it is either not opened in write mode, is marked as a read-only file on the disk or writes are being blocked by another user
`UTFAT_DIRECTORY_AREA_EXHAUSTED` - the end of the FAT space was reached and no valid clusters found
`UTFAT_DISK_WRITE_PROTECTED` – the SD card/memory stick is write-protected

A write to a file is made at its present file position, whereby the position is at the start of the file when it is first opened (or at end when using the open flag `UTFAT_APPEND`). This generally allows content to be overwritten. If additional content is to be added, the file position should first be change accordingly using the `utSeek()` command (unless the `UTFAT_APPEND` flag was used to open the file). When additional data is written over the end of the present file the file's length will be increased accordingly. After a write the amount of data written is held in `ptr_utFile->usLastReadWriteLength`.

Should the file be opened by other users in managed mode the file objects of the other users will be updated in case of changes to the file size. *If data caching is being used the write may be set to the data cache only but all other users will access the data cache when reading from the corresponding sector and so always be synchronised with the written data even when it has not yet been committed to disk*.

If the file is not opened for write, or is protected by another user, any attempted write to it will fail. The same is true when the file is marked as a read-only file on the disk or if the disk is generally write protected.

To avoid multiple users writing a single file it is advisable to open it as a managed file, which will stop a second user from being able to open it for write.

```
if (utWriteFile(&utFile, "Test Content", 12) != UTFAT_SUCCESS) {
    fnDebugMsg("write failed");
}
else {
    fnDebugMsg("New file length = ");
    fnDebugDec(utFile.ulFileSize, 0);
}
fnDebugMsg("\r\n");
```

Note that the file object's position pointer is modified by reads and writes. There is one pointer shared by both functions.

11.8. utRenameFile()

```
extern int utRenameFile(const CHAR *ptrFilePath, UTFILE *ptr_utFile);
```

This function allows a file or a directory to be renamed.

The routine returns UTFAT_SUCCESS if the file or directory was successfully deleted.

Errors can be:
UTFAT_DIR_NOT_EMPTY – a directory could not be deleted because it is not empty
UTFAT_PATH_NOT_FOUND – either original file/directory not found or new path not found
UTFAT_DISK_READ_ERROR - error occurred while trying to read a sector from the disk
UTFAT_FILE_NOT_WRITEABLE - the file cannot be written because it is either not opened in write mode, is marked as a read-only file on the disk or writes are being blocked by another user
UTFAT_DIRECTORY_AREA_EXHAUSTED - the end of the FAT space was reached and no valid clusters found
UTFAT_DISK_WRITE_PROTECTED – the SD card is write-protected
LFN_RENAME_NOT_POSSIBLE – a LFN has been detected but support for LFN writing is not oenabled

Files and directories are essentially treated the same since both have a file entry.

Example of renaming a file:

```
if (utRenameFile("dir2/file4.txt", "dir2/file5.txt") != UTFAT_SUCCESS) {
    fnDebugMsg("Rename failed\r\n");
}
```

Example of renaming a directory:

```
if (utRenameFile("dir3", "dir4") != UTFAT_SUCCESS) {
    fnDebugMsg("Rename failed\r\n");
}
```

11.9. utDeleteFile()

```
extern int utDeleteFile(const CHAR *ptrFilePath, UTDIRECTORY *ptrDirObject);
```

This function allows a file or a directory to be deleted.
If a file is deleted, the file entry is marked as deleted and its content cluster chain is freed in the FAT table. The file's data content is not deleted from the disk and it may be possible to undelete the file later.

A directory can only be deleted when it is empty.

The routine returns `UTFAT_SUCCESS` if the file or directory was successfully deleted.

Errors can be:
`UTFAT_DIR_NOT_EMPTY` – a directory could not be deleted because it is not empty
`UTFAT_PATH_NOT_FOUND` – file or directory not found
`UTFAT_DISK_READ_ERROR` - error occurred while trying to read a sector from the disk
`UTFAT_FILE_NOT_WRITEABLE` - the file cannot be written because it is either not opened in write mode, is marked as a read-only file on the disk or writes are being blocked by another user
`UTFAT_DIRECTORY_AREA_EXHAUSTED` - the end of the FAT space was reached and no valid clusters found
`UTFAT_DISK_WRITE_PROTECTED` – the SD card is write-protected

Files and directories are essentially treated the same since both have a file entry. Directories must however be empty before they can be deleted.

Example of deleting a file:

```
if (utDeleteFile("dir1/file1.txt", ptr_utDirectory) != UTFAT_SUCCESS) {
    fnDebugMsg("Delete failed\r\n");
}
```

Example of deleting a directory:

```
Int iResult = utDeleteFile("dir3", ptr_utDirectory);
if (iResult != UTFAT_SUCCESS) {
    if (iResult == UTFAT_DIR_NOT_EMPTY) {
        fnDebugMsg("Directory can only be deleted when empty!\r\n");
    }
    else {
        fnDebugMsg("Delete failed\r\n");
    }
}
```

11.10. utSafeDeleteFile()

```
extern int utSafeDeleteFile(const CHAR *ptrFilePath, UTDIRECTORY *ptrDirObject);
```

This function allows a file or a directory to be deleted without leaving information on the disk.
If a file is deleted the file entry is completely destroyed, the content's content cluster chain is freed in the FAT table and the original file data content is written with zeroes.

A directory can only be deleted when it is empty.

The routine returns `UTFAT_SUCCESS` if the file or directory was successfully deleted.

Errors can be:
`UTFAT_DIR_NOT_EMPTY` – a directory could not be deleted because it is not empty
`UTFAT_PATH_NOT_FOUND` – file or directory not found
`UTFAT_DISK_READ_ERROR` - error occurred while trying to read a sector from the disk
`UTFAT_FILE_NOT_WRITEABLE` - the file cannot be written because it is either not opened in write mode, is marked as a read-only file on the disk or writes are being blocked by another user
`UTFAT_DIRECTORY_AREA_EXHAUSTED` - the end of the FAT space was reached and no valid clusters found
`UTFAT_DISK_WRITE_PROTECTED` – the SD card is write-protected

Files and directories are essentially treated the same since both have a file entry. Directories must however be empty before they can be deleted.

Example of safely deleting a file:

```
if (utSafeDeleteFile("dir1/file1.txt", ptr_utDirectory) != UTFAT_SUCCESS) {
    fnDebugMsg("Delete failed\r\n");
}
```

Example of safely deleting a directory:

```
Int iResult = utSafeDeleteFile("dir3", ptr_utDirectory);
if (iResult != UTFAT_SUCCESS) {
    if (iResult == UTFAT_DIR_NOT_EMPTY) {
        fnDebugMsg("Directory can only be deleted when empty!\r\n");
    }
    else {
        fnDebugMsg("Delete failed\r\n");
    }
}
```

Beware that the complete existing file content is deleted during the call, which may require a long time in case the original file was large. If the delete time is an issue it may be better for the user to write the present content to a deleted state using a managed write method (background task), truncate the file to zero length and then finally call the safe delete function to remove the remaining file object itself.

11.11. utReadFile()

```
extern int utReadFile(UTFILE *ptr_utFile,
                      unsigned char *ptrBuffer,
                      unsigned short usLength);
```

This function allows content (binary) to be read from the file referenced by `ptr_utFile`. The amount of data `usLength`, or the amount that can be read if the file end is reached, is copied to the space `ptrBuffer`.

The routine returns `UTFAT_SUCCESS` if the data was successfully read.
The amount of data read is contained in `ptr_utFile->usLastReadWriteLength`

Errors can be:
`UTFAT_DISK_READ_ERROR` - error occurred while trying to read a sector from the disk
`UTFAT_FILE_NOT_READABLE` - the file is not opened in read mode
`UTFAT_DIRECTORY_AREA_EXHAUSTED` - the end of the FAT space was reached and no valid clusters found

When the file is opened its pointer position is generally at the start of the file and the first read returns data from the start of the file unless the pointer is first modified using `utSeek()`. Each subsequent read increments the pointer to the end of the read block so that multiple reads work through the file from its start to its end. If the end of the file was reached while reading, the amount of data that could be read may be less than `usLength`. The value actually read is contained in `ptr_utFile->usLastReadWriteLength` and will be 0 if the file is empty or the file pointer is already at the end of the file.

Multiple users can read from a file. If the file is opened as a managed file the file object will automatically be updated when it is changed by a write by another user (for example when its size is increased).

```
unsigned char ucTemp[256];      // temp buffer to retrieve a block of data from the file

if (utReadFile(&utFile, ucTemp, sizeof(ucTemp)) != UTFAT_SUCCESS) {
     fnDebugMsg("READ ERROR occured\r\n");
}
else {
    fnDebugMsg("Length read = ");
    fnDebugDec(utFile.usLastReadWriteLength, 0);
    if (utFile.usLastReadWriteLength < sizeof(ucTemp)) {
        fnDebugMsg(" End of file reached");
    }
    fnDebugMsg("\r\n");
}
```

Note that the file object's position pointer is modified by reads and writes. There is one pointer shared by both functions.

11.12. utCloseFile()

```
extern int utCloseFile(UTFILE *ptr_utFile);
```

This function is only absolutely necessary when the file was opened in managed file mode so that the file is no longer owned by the user (which can block other users from writing it).
The call frees the file from the managed file list (if managed file mode is used) and clears the `UTFILE` struct that is pointed to by `ptr_utFile`.
Files that have been opened with the attribute `UTFAT_COMMIT_FILE_ON_CLOSE` have changes in the file object committed to disk only when the file is closed.

Example:

```
UTFILE utFile = {0};
ptr_utFile.ownerTask = OWN_TASK;
utOpenFile("dir1/file6.txt" &utFile, ptr_utDirectory,
       (UTFAT_OPEN_FOR_WRITE | UTFAT_CREATE | UTFAT_TRUNCATE | UTFAT_MANAGED_MODE));
...

utCloseFile(&utFile);               // free from managed file list and clear utFile
```

11.13. utMakeDirectory()

```
extern int  utMakeDirectory(const CHAR *ptrDirPath, UTDIRECTORY *ptrDirObject);
```

This function allows a new directory to be created.

The routine returns `UTFAT_SUCCESS` if the directory was successfully created.

Errors can be:
`UTFAT_PATH_NOT_FOUND` – error in path
`UTFAT_INVALID_NAME` – invalid director name
`UTFAT_DISK_READ_ERROR` – low level error
`UTFAT_DIRECTORY_EXISTS_ALREADY` – directory already exists
`UTFAT_DIRECTORY_AREA_EXHAUSTED` - the end of the FAT space was reached and no valid clusters found
`UTFAT_DISK_WRITE_PROTECTED` - the SD card/memory stick is write-protected

Creates a directory in the present directory, or in a referenced directory.

```
if (utMakeDirectory("dir1/new_dir", ptr_utDirectory) != UTFAT_SUCCESS) {
    fnDebugMsg("Make dir failed\r\n");
}
```

11.14. utLocateDirectory()

```
extern int utLocateDirectory(const CHAR *ptrDirPath, UTLISTDIRECTORY *ptrListDirectory);
```

This function is used to fill out a list to a directory which can subsequently be used to work with files and directories in the directory.

The routine returns `UTFAT_SUCCESS` if the data was successfully written.

Errors can be:
`UTFAT_DIRECTORY_OBJECT_MISSING` – no directory object is specified
`UTFAT_DISK_NOT_READY` - disk not ready for use - not formatted or not mounted
`UTFAT_PATH_NOT_FOUND` - the referenced directory path could not be found
`UTFAT_DISK_READ_ERROR` - error occurred while trying to read a sector from the disk
`UTFAT_PATH_IS_FILE` - the referenced object is a file and not a directory

The user passes a list directory object which is filled by the function with information about the directory location which can subsequently be used by further operations. This is used specifically for working together with the `utListDir()` function which allows simple directory content listing.

```
UTLISTDIRECTORY utListDirectory;                    // list directory object for a single user

utListDirectory.ptr_utDirObject = ptr_utDirectory; // reference the list directory to main
                                                      directory object
if (utLocateDirectory("dir", &utListDirectory) < UTFAT_SUCCESS) {
                                                   // open a list referenced to the main directory
    fnDebugMsg("Invalid directory\r\n");
}
```

11.15. utListDir()

```
extern int utListDir(UTLISTDIRECTORY *ptr_utDirectory, FILE_LISTING *ptrFileLists);
```

This function is used to move through the content of a directory. It uses a passed directory list and fills out the next file/directory in the directory on each call.

The list directory is originally filled by using `utLocateDirectory()`.

The routine returns `UTFAT_NO_MORE_LISTING_ITEMS_FOUND` when all directory entries have been worked through or `UTFAT_SUCCESS` it successfully fills out an entry.

Errors can be:
`UTFAT_DISK_NOT_READY` - disk not ready for use - not formatted or not mounted
`UTFAT_DIRECTORY_AREA_EXHAUSTED` - the end of the FAT space was reached and no valid clusters found
`UTFAT_NO_MORE_LISING_SPACE` – no more items can be filled out due to lack of buffer space (this may not be a serious error)

`utLocateDirectory()` is typically used for working through items in a directory (files and sub-directories) and can be used to simply list directory content as shown by the following example:

```
UTLISTDIRECTORY utListDirectory;
FILE_LISTING fileList = {0};
CHAR cBuffer[MAX_UTFAT_FILE_NAME + DOS_STYLE_LIST_ENTRY_LENGTH];
                                                // temporary string buffer for listing
fileList.usMaxItems = 1;                        // get just one item at a time
fileList.ptrBuffer = cBuffer;
fileList.usBufferLength = sizeof(cBuffer);
fileList.ucStyle = DOS_TYPE_LISTING;            // format as required for DOS style listing
                                                // (FTP uses FTP_STYLE_LIST_ENTRY_LENGTH)
utListDirectory.ptr_utDirObject = ptr_utDirectory;
utLocateDirectory("dir1/dir2", &utListDirectory); // prepare the list directory

While (utListDir(&utListDirectory, &fileList) != UTFAT_NO_MORE_LISTING_ITEMS_FOUND) {
    // fileList contains information about the next file/directory in the present directory
    // as well as a string formatted for output
    fnWrite(DebugHandle, (unsigned char *)cBuffer, (QUEUE_TRANSFER)fileList.usStringLength);
}
```

See `utLocateDirectory()` for details about preparing `utListDirectory` for use by this example.

The entries filled out by the function are shown in red – the parameters passed are blue:

```
typedef struct stFILE_LISTING
{
    CHAR *ptrBuffer;                    // pointer to character buffer
    unsigned long  ulFileSizes;         // sum of the total file sizes in this listing
    unsigned short usBufferLength;      // length available in character buffer
    unsigned short usStringLength;      // length added to character buffer
    unsigned short usMaxItems;          // maximum items to be treated in this pass
    unsigned short usItemsReturned;     // the number of items treated in this pass
    unsigned short usDirectoryCount;    // the number of directories treated in this pass
    unsigned short usFileCount;         // the number of files treated in this pass
    unsigned char  ucStyle;             // the formatting style to be used
} FILE_LISTING;
```

11.16. fnGetDiskInfo()

```
extern const UTDISK *fnGetDiskInfo(unsigned char ucDisk);
```

This function is used to get a pointer to the disk

The user can collect a pointer to main information about the disk.

```
UTDISK *ptrDiskInfo = fnGetDiskInfo(DISK_D);
```

The UTDISK struct contains the following items:

```
typedef struct stUTDISK
{
    unsigned long  ulPresentSector;   // the present sector being used by the disk
    unsigned long  ulDirectoryBase;   // the first cluster in the root directory (usually 2)
    unsigned long  ulLogicalBaseAddress;  // first cluster containing data
    unsigned long  ulVirtualBaseAddress;  // virtual cluster starting address,
                                             compensating unused clusters
    unsigned long  ulSD_sectors;       // physical sectors on the device
    unsigned char  *ptrSectorData;     // pointer to a buffer containing a copy of the sector
                                          data
    unsigned short usDiskFlags;        // flags indicating the status of the disk
    UTFAT          utFAT;              // FAT information concerning the data content
    FILEINFO       utFileInfo;         // file information used by FAT32
    unsigned char  ucDriveNumber;      // the drive number of this disk
    CHAR           cVolumeLabel[11];   // the volume's label
} UTDISK;
```

The entry usDiskFlags is an important item since it allows the application to know the exact state of the disk. Its flag are shown below:

```
FSINFO_VALID                    - the disk has a valid info block which can be used to
                                  accelerate some calculations
DISK_UNFORMATTED                - disk detected but its content is not formatted
WRITE_PROTECTED_SD_CARD         - the SD card has write protection active and so no write
                                  operations are allowed
DISK_FORMATTED                  - the disk has been detected and is formatted
HIGH_CAPACITY_SD_CARD           - the disk is of high capacity type
DISK_MOUNTED                    - the disk has been mounted and so is ready for use
DISK_NOT_PRESENT                - a check of the disk failed to identify its presence
DISK_TYPE_NOT_SUPPORTED         - unsupported disk type detected
DISK_FORMAT_FULL                - set all content to 0x00 rather than just all FAT
DISK_FORMAT_FAT12               - FAT12 format rather than FAT32
DISK_FORMAT_FAT16               - FAT16 format rather than FAT32
DISK_FORMAT_EXFAT               - exFAT format rather than FAT32
DISK_TEST_MODE                  - special flag to control testing (development tests)
```

User code should never change any values in the UTDISK struct!!

11.17. utFreeClusters()

```
extern int  utFreeClusters(unsigned char ucDisk, UTASK_TASK owner_task);
```

This function is used start a count of the free clusters available on the SD card/memory stick. It is only needed when the there is no valid info-block, which otherwise contains this value.

When the free cluster count has been obtained the interrupt event UTFAT_OPERATION_COMPLETED is sent to the requesting task, which can then read the (updated) value from the UTDISK pointer.

The routine returns UTFAT_SUCCESS if it successfully starts the free cluster count.

Errors can be:
MISSING_USER_TASK_REFERENCE – no task reference was passed

Example of checking whether the free cluster count is valid and starting a count if necessary:

```
UTDISK *ptrDiskInfo = fnGetDiskInfo(DISK_D);
if ((ptrDiskInfo->usDiskFlags & FSINFO_VALID) &&
    (ptrDiskInfo->utFileInfo.ulFreeClusterCount != 0xffffffff)) {
    fnDebugMsg("Free cluster count = ");          // free cluster value is known
    fnDebugDec(ptrDiskInfo->utFileInfo.ulFreeClusterCount, WITH_CR_LF);
}
else {
    iFATstalled = STALL_COUNTING_CLUSTERS;       // mark that we are expecting a result when
                                                    completed
    utFreeClusters(DISK_D, OWN_TASK);            // start count of free clusters - the result
                                                    will be displayed on completion
}
```

If the cluster count was started the task entered as reference will be woken with the interrupt event UTFAT_OPERATION_COMPLETED. The task can then use the value in ptrDiskInfo->utFileInfo.ulFreeClusterCount as present free cluster count, even if there is no valid info-block.

It is to be noted that a free cluster count can take some time on a large SD card/memory stick since all of the FAT has to be read. This is the reason for allowing the mass-storage task to do this as a background activity.

11.18. utFormat()

```
extern int  utFormat(const unsigned char ucDrive,
                     const CHAR *cVolumeLabel,
                     unsigned char ucFlags);
```

This function is used start formatting an SD card/memory stick. It accepts an optional volume label and the formatting performed depends on the flags passed.

The flag combination allows formatting as FAT16, FAT12, exFAT or FAT32 as well as re-formatting an already formatted card.

The routine returns `UTFAT_SUCCESS` if it successfully starts the formatting/re-formatting.

Errors can be:
`UTFAT_DISK_NOT_READY` – an attempt was made to format formatted disk
(use `UTFAT_REFORMAT` flag)
`UTFAT_DISK_WRITE_PROTECTED` – write protected disk can't be formatted/re-formatted

Example of starting the format of a non-formatted an SD card:

```
if (utFormat(DISK_D, "UTFAT_DISK", (UTFAT_FORMAT | UTFAT_FORMAT_32)) != UTFAT_SUCCESS) {
    fnDebugMsg("not possible\r\n");
}
fnDebugMsg("in progress - please wait...\r\n");
```

The formatting flags are shown below, whereby it is to note that FAT32 is the default and is used when no flag is specified:

```
#define UTFAT_FORMAT                // format only non-formatted disk (default)
#define UTFAT_REFORMAT              // reformat already formatted disk
#define UTFAT_FORMAT_12             // format as FAT16 rather than FAT32
#define UTFAT_FORMAT_16             // format as FAT16 rather than FAT32
#define UTFAT_FORMAT_32             // format as FAT32 (default if nothing specified)
#define UTFAT_FORMAT_EXFAT          // format as exFAT rather than FAT32
#define UTFAT_FULL_FORMAT           // perform full format - including deleting
                                       existing cluster content
```

11.19. utReadSector()

```
extern int  fnReadSector(unsigned char ucDisk,
                         unsigned char *ptrBuffer,
                         unsigned long ulSectorNumber);
```

This function is used to read a sector from the SD card/memory stick. It is a low-level command used normally only for testing since it bypasses the µtFAT. It can also be used when the SD card/memory stick is not formatted or its format is otherwise invalid.

The routine returns `UTFAT_SUCCESS` if the read was successful

Errors can be:
`ERROR_CARD_TIMEOUT` – timeout error reading from card
`UTFAT_DISK_READ_ERROR` - error occurred while trying to read a sector from the disk

When `ptrBuffer` is set to an array of 512 bytes (this is needed to be able to accept the complete sector content) the data read from the sector is copied directly to this buffer.

If a `ptrBuffer` is set to 0 the read is performed into the disk's sector buffer (see `ptrSectorData` in `UTDISK` which can be accessed via the pointer returned by `fnGetDiskInfo()`.

11.20. utWriteSector()

```
extern int  fnWriteSector(unsigned char ucDisk,
                          unsigned char *ptrBuffer,
                          unsigned long ulSectorNumber);
```

This function is used to write a sector on the SD card/memory stick directly with user data. It is a low-level command used normally only for testing since it bypasses the µtFAT and can cause FAT or data corruption. It can also be used when the SD card/memory stick is not formatted.

The routine returns `UTFAT_SUCCESS` if the write was successful

Errors can be:
`ERROR_CARD_TIMEOUT` – timeout error reading from card
`UTFAT_DISK_WRITE_ERROR` - error occurred while trying to write a sector to the disk

`ptrBuffer` must be set to an array of 512 bytes which contains the data to be written to the sector.

12. µtFAT File Management

To do...

In the meantime please us the µtFAT forum for questions and answers:
http://www.utasker.com/forum/index.php?board=10.0

13. Long File Name Support

Support for reading long file names (LFN) reading is activated by the define `UTFAT_LFN_READ`. This means that files created with LFN can be accessed and displayed by that name. Deleting LFN is also possible but the delete only deletes the short file name part (the part compatible with FAT systems that only see the LFN as "`FILENA~1.EXT`")

When the define `UTFAT_LFN_DELETE` is enabled deleting of the LFN part is also performed to avoid leaving file name entries on the disk. *Details about leaving undeleted LFN entries is discussed later*.

Deleting LFN files and renaming them to short file names is possible when `UTFAT_LFN_READ` is enabled. Renaming LFN files to different LFNs is only possible when the define `UTFAT_LFN_WRITE` is enabled. When an attempt is made to rename a file which is in LFN format (it is possible that also file names which look to fit in the 8:3 format are actually stored in LFN format) to a file that requires LFN results in the error `LFN_RENAME_NOT_POSSIBLE` when this is not supported.

LFN renaming and writing is enabled by activating the project define `UTFAT_LFN_WRITE`. This achieves full compatibility with LFN and SFN FAT16/32 file systems. Since LFN is protected by patents the use of the full features may require a licensing agreement with Microsoft so the full feature should be used only when the legal situation is clear. With the define `UTFAT_LFN_WRITE_PATCH` a workaround is activated which limits compatibility with older systems that can only work with SFN but is believed to avoid the patent issue. This is the approach taken by Linux VFAT as detailed by the author at https://lkml.org/lkml/2009/6/26/314 . Patent issues are never fully clear but if the Linux patch sufficiently avoids infringement it is expected that this option will also stand up to the same scrutiny. In any case, the user has final responsibility for what options are used in a particular project.

13.1. Brief History of Long File Names

There are various articles about the history of LFN so this section is not going to repeat too much detail but rather point out a few things that should be understood for compatibility with systems supporting only LFN, only short file names (SFN) or both.

SNF has already been seen in operation whereby a single directory or file entry occupies a single directory entry (`DIR_ENTRY_STRUCTURE_FAT32`), which is 32 bytes in size. There are 11 bytes in the entry for the name and it is limited due to this to the well-known 8:3 format, where the last 3 bytes are understood as the extension and the first 8 (or less) as the name. FAT systems that don't understand LFN (like pre-Windows 95 software) must still be able to work with the directories and files created by LFN and so the following trick is used to ensure this basic compatibility:

> A directory attribute entry (`DIR_ENTRY_STRUCTURE_FAT32.DIR_Attr`) with a value of 0x0f is an invalid entry for a system that understands only SFN. This value would mean that it is a "hidden, read-only system volume ID", which is invalid and ignored, meaning that the rest of the entry space can be used to save something else – in this case the LFN part of the entry. Usually a LFN entry is constructed from a number of entries following each other to allow file names of up to 255 UTF-16 characters to be used.

For compatibility also a short file name entry is added when a directory or file is created, which follows the LFN entries. A system understanding only SFN will therefore simply skip all of the LFN parts and use just the following SFN.

Note that the fact that LFN actually uses a LFN part and a SFN part is the central part of the LFN patent.

Since a LFN will normally not fit into the name space in the SFN entry the actual file name typically has to be shortened and this is known as the short file name alias of the long file name. A LFN called "`My +Long File A.txtFile`" would be saved with the SNF alias "`MY_LON~1.TXT`". A second LFN called "`My +Long File B.txtFile`" in the same directory would be saved with the SFN alias "`MY_LON~2.TXT`", etc.

The rules for the alias are fairly straight forward:

- The extension is cut down to the maximum 3 position (`txtFile` was cut down to `txt`).
- The first 6 letters (excluding spaces) are used.
- Any unsupported characters (`+,;=[]` are not allowed by SFN but are allowed by LFN) are replaced by `_`.
- All characters are converted to upper-case.
- The final two characters in the name are set with `~1` as long as this file doesn't already exist in the same directory. If it does it will try `~2`, etc. until it finds a name that doesn't conflict.

The result is a solution which allows files created by SFN systems to be used by SFN and LFN systems. Files created by LFN systems can be used (although with an alias name) by SFN systems. This solution is however not without its pitfalls:

- Older programs that cleaned up disks would recognise the LFN parts of files as invalid and happily clean them up, hence losing the LFN information.
- Saving a LFN file under a new name with a SFN system will lose its original LFN information.

- The SFN alias rules theoretically only allow 9 files with ‘similar’ LFNs to be used – practically PCs will start adding random characters to fill the 3 characters before the ~.
- Systems that don’t understand LFN will delete only the SFN part of the file’s LFN entry, leaving the LFN parts there. This may lose a small amount of space but could be cleaned up later by a LFN aware system.
  SFN systems may reuse deleted SFN entries at the end of the LFN part which then doesn’t match the LFN – *to avoid LFN systems from using such names the LFN entry includes a check sum of the original LFN alias, as is described later*.
- As will be seen below, the LFN needs to be stored in a linear area large enough to hold the complete name. This makes reuse of deleted directory entries more complicated since a hole of adequate size needs to be found to be able to hold the new name. The result is that the directory area may grow larger than expected due to such holes (deleted entry space that is not large enough to reuse for a new LFN).
- When LFN operation was first used there were some problems with older SFN software that could mistake LFN entries in the root directory as volume IDs and so displayed the volume ID incorrectly.
- Initially there was a risk of two long file named files saved with different names in two different folders being give the same SFN aliases and then being saved in the wrong folder, thus overwriting one of the original files with the content of a different one. Users had to be very careful.
- Although there were some initial difficulties as LFN was first introduced in Windows 95 in regard to compatibility between old software and new file, and new software and old files these are mostly irrelevant nowadays since such systems are legacy.

13.2. LFN Entries

Here we take a look as the LFN entry "`My +Long File E.txtFile`", which is made up of three directory entries following each other – *they are displayed in little-endian long-word format with each line showing a single directory entry of 32 bytes*:

```
0x45002042 0x74002e00 0x0f007800 0x00740500 0x00690046 0x0065006c 0x00000000 0xffffffff
0x79004d01 0x2b002000 0x0f004c00 0x006f0500 0x0067006e 0x00460020 0x00000069 0x0065006c
0x4c5f594d 0x317e4e4f 0x20545854 0xad43a300 0x40654065 0xacf80000 0x00004065 0x00000000
```

„`MY_LON~1.TXT`“

Looking at the directory attributes (with red background) the two LFN entries (0x0f) are clearly visible, followed by a normal file entry (0x20). The entry with SFN alias together with the file data/time, length and starting cluster holds the main information about the file itself.

Each LFN entry (there are two in this example) can be represented as follows:

```
typedef struct stLFN_ENTRY_STRUCTURE_FAT32
{
    unsigned char LFN_EntryNumber;        // entry number starting from last - 0x40 is
                                             always set in first entry and the value
                                             decrements until 1
    unsigned char LFN_Name_0;             // first letter
    unsigned char LFN_Name_0_extension;   // first letter extension - is always 0 in English
                                             character set
    unsigned char LFN_Name_1;             // second letter
    unsigned char LFN_Name_1_extension;   // second letter extension
    unsigned char LFN_Name_2;             // third letter
    unsigned char LFN_Name_2_extension;   // third letter extension
    unsigned char LFN_Name_3;             // fourth letter
    unsigned char LFN_Name_3_extension;   // fourth letter extension
    unsigned char LFN_Name_4;             // fifth letter
    unsigned char LFN_Name_4_extension;   // fifth letter extension
    unsigned char LFN_Attribute;          // always 0x0f
    unsigned char LFN_Zero0;              // always zero
    unsigned char LFN_Checksum;           // check sum
    unsigned char LFN_Name_5;             // sixth letter
    unsigned char LFN_Name_5_extension;   // sixth letter extension
    unsigned char LFN_Name_6;             // seventh letter
    unsigned char LFN_Name_6_extension;   // seventh letter extension
    unsigned char LFN_Name_7;             // eighth letter
    unsigned char LFN_Name_7_extension;   // eighth letter extension
    unsigned char LFN_Name_8;             // ninth letter
    unsigned char LFN_Name_8_extension;   // ninth letter extension
    unsigned char LFN_Name_9;             // tenth letter
    unsigned char LFN_Name_9_extension;   // tenth letter extension
    unsigned char LFN_Name_10;            // eleventh letter
    unsigned char LFN_Name_10_extension;  // eleventh letter extension
    unsigned char LFN_Zero1;              // always zero
    unsigned char LFN_Zero2;              // always zero
    unsigned char LFN_Name_11;            // twelfth letter
    unsigned char LFN_Name_11_extension;  // twelfth letter extension
    unsigned char LFN_Name_12;            // thirteenth letter
    unsigned char LFN_Name_12_extension;  // thirteenth letter extension
} LFN_ENTRY_STRUCTURE_FAT32;
```

```
0x45002042 0x74002e00 0x0f007800 0x00740500 0x00690046 0x0065006c 0x00000000 0xffffffff
0x79004d01 0x2b002000 0x0f004c00 0x006f0500 0x0067006e 0x00460020 0x00000069 0x0065006c
```

Looking only at the two LFN entries in the example it is seen that the entry number always starts with the bit 0x40 set. In this case it is 0x42 because the LFN part consists of 2 entries. The second entry is then numbered one less that the first (0x01), without the 0x40 bit set. This makes it easy to see which of the entries is the first and to verify that they follow each

other correctly. After the entry number 1, either 0x41 when there is only one LFN entry, or 0x01 when it is the final entry in the chain, the SFN alias is to be expected.

There is space for up to 13 characters in each entry. The characters are in UTF-16 format to allow a large variety of character sets to be used. When only the English character set is used the extension byte of each character is always 0.

Although it would be possible to have a maximum 0x3f (63) entries, each with up to 13 characters, giving a maximum LFN length of 819 (including a final null-terminator), a maximum of 20 LFN entries is allowed, and a maximum of 255 characters.

There are 3 bytes that have to be set to 0x00 in each entry.

The check sum (0x05 in the example) is the same in each on the LFN entries because it is calculated over the SFN alias. If the check sum doesn't match the SFN alias found after the LFN part it probably means that the SFN was deleted by a system not understanding LFN (thus leaving the LFN part intact) and reused the deleted space at some point for a new file name. The new SFN entry then is not related to the LFN part and the LFN part should be ignored or could even be deleted by a system understanding LFN. The LFN system this needs to check that the checksum value in the LFN part indeed matches with the value calculated over the SFN alias following it.

When reading a valid LFN the individual characters are extracted from the chain of LFN entries. This takes place in a reverse order, starting in the first entry. At the end of the entry there are some 0xffff characters which represent padding because the name doesn't completely fill the entry. Then, reading from the end to the beginning, there is a 0x0000, which is the null-terminator at the end of the file name. Continuing towards the start of the entry and just considering the standard part of the UCF-16 characters the name of the file can be collected in the reverse order:

```
0x65 ,e'
0x6c ,l'
0x69 ,i'
0x46 ,F'
0x74 ,t'
0x78 ,x'
0x74 ,t'
0x2e ,.'
0x45 'E'
0x20 , ,
0x65 ,e'
0x6c ,l'
0x69 ,i'
0x46 ,F'
0x20 , ,
0x67 ,g'
0x6e ,n'
0x6f ,o'
0x4c ,L'
0x2b ,+'
0x20 , ,
0x79 ,y'
0x4d ,M'
```

13.3. Deleting LFN Entries

A SFN system can delete LFN names but such a system only deletes the SFN alias and file information, meaning that the LFN part is left in the cluster. This is generally not a problem since the space used up is not large on today's SD cards. A LFN aware system could also detect such lost LFN entry chains and clean them up if required.

When the define `UTFAT_LFN_DELETE` is enabled the µtFAT deletes the LFN part of the LFN entry clusters when deleting a file, even when it is not configured to support writing LFNs.

The delete process means not just deleting the entry containing the SFN alias but also deleting all previous LFN parts of the entry before it. To delete these, the first byte of each entry is simply set to the free value of `0xe5`.

The only complication in this process occurs when previous parts of the LFN reside in a different cluster to the cluster that the SFN alias part is in. The complication arises from the fact that clusters cannot be searched in a reverse direction. However this is quite simply overcome by remembering the sector in which the first entry in the LFN entry chain was found in when the LFN was handled (the LFN is always handled by running through its entry chain). Furthermore, because there will never be more than 20 such entries there can only be one possible cluster change in the process: 20 entries of 64 bytes requires a linear space of 1280 bytes, which is smaller than the size of a cluster used in such systems. This means that if a cluster boundary is encountered when deleting a LFN entry chain in the reverse direction the next sector to use will be at the end of the cluster in which the first entry was in.

13.4. Renaming LFN Entries

Renaming long file names to short file names is quite simple because it means overwriting the SFN alias with the new name. A LFN system will recognise that the non-deleted LFN part is no longer associated with the SFN due to the fact that its check sum is no longer valid. Both LFN and SFN systems will see only the new SFN.

When `UTFAT_LFN_DELETE` is enabled also the LFN part is deleted in an analogue manner to the file deletion as described in the previous section.

If LFN support is not enabled for writing an attempt to rename a file to a LFN will result in the error `LFN_RENAME_NOT_POSSIBLE`.

When LFN writing is enabled (`UTFAT_LFN_WRITE`), renaming existing files to LFNs is essentially equivalent to creating a new LFN entry (see next section) but keeping the original file content. The only thing that is slightly different is the fact that it may not be possible to rename using the original entry location. If the size of the file name however needs to occupy more entries than the original the complete location may need to be changed and the complete original entry fully deleted. If possible the original LFN entries will be reused and extended entries added to the end of it. If this is not possible because there is no space following the original entry a completely new space must be found that is large enough for the new name and the original file entry deleted. It is also possible that the allocation of additional clusters becomes necessary for the directory increased directory space.

The following list illustrates the various cases that can exist when renaming a file in a system supporting both SFN and LFN with an overview of the technique used. It is worth noting that before a rename the directory needs to be searched to ensure that there is no file already with the new name – this search is useful for collecting information that allows all possible

cases to be simply resolved [the location of the end of the directory is known, the location of deleted entries in the present directory are can be collected]:

1. *Renaming SFN to SFN* - The SFN can be modified directly.
2. *SFN renamed to a* LFN - The original SFN entry is not large enough for the new LFN and so the new LFN needs to be relocated and the original SFN information transferred to the SFN alias of the new LFN. The original SFN is marked as deleted. If there is deleted space large enough to accept the new LFN it is used, or the new LFN is located at the end of the directory.
3. *LFN renamed to a SFN* - The original SFN alias is used as new SFN while the original LFN information is marked as deleted.
4. *LFN renamed to a LFN with equal number of LFN entries* - The original SFN alias is used as new SFN alias and the LFN entries reused.
5. *LFN renamed to a LFN with less number of LFN entries* - The original SFN alias is used as new SFN alias. The first LFN entry/entries are deleted and the rest used for the new LFN entry/entries.
6. *LFN renamed to a LFN requiring more LFN entries that the original* name – The LFN entries and SFN alias are marked as deleted. A new LFN is generated either in deleted space adequately large to accept it or else at the end of the directory. The original SFN alias information is transferred to the new SFN alias.

13.5. Creating new LFN Entries

New LFNs can be created when its support is defined by `UTFAT_LFN_WRITE`

New LFNs (or files renamed to LFNs as discussed in the previous section) are handled by the rules introduced in this chapter. This means that a SNF alias is generated for the file which doesn't conflict with any other SFN aliases in the same directory and checksum for that alias is calculated.

The number of directory entries to hold the LFN part plus the SFN alias (with the actual file data information) is calculated and this space is identified in the directory. Either deleted entries are used, if there are enough deleted linear entries found, or else new directory entry space is identified at the end of the directory; this may involve extending the directory with additional clusters as is the standard case when new data is added to a directory.

The LFN parts are written containing the LFN information as details in this chapters and the SFN alias, containing the file's data information, is written.

If the define `UTFAT_LFN_WRITE_PATCH` is enabled the SFN alias is not written using the standard technique. This means that there is limited backward compatibility with systems that only understand SFNs but this technique seems to avoid infringement of the LFN patents which essentially covers the technique of saving file names as both short and long names at the same time. Rather than devise a SFN alias according to the standard rules, eg. "`SHORTF~1.txt`" the short file name is constructed using random characters from the invalid file name character set. The result is that the short file names, and the file's data information, can still be used to verify that the entry belongs to the LFN part but old systems that can only read SFNs will tend not to see the files at all since their names are invalid. The short file name part can therefore not be considered a valid name and so

The technique therefore has some limits to its backward compatibility for legacy systems but is usable in the majority of modern environments where such requirements make little sense. It is understood that the invalid SFN alias doesn't disturb modern file systems that are only interested in the LFN part and the data that belongs to it but there are reports of the technique being able to cause some older systems which do use LFN to crash – specifically a Windows XP bug has been mentioned that causes a blue screen when attempting to read such files. The algorithm chosen for the generation of the SFN alias minimises the triggering of such a bug through its choice of character values. Windows Vista and Windows 7 are not know to have this bug.

The algorithm defined by the original author can be found at
https://lkml.org/lkml/2009/6/26/313

If a new file or a renamed file can be saved as a SFN this is always done in preference to a LFN.

14. Data Caching and Speed Optimisations

The FAT interface includes a single sector buffer. Each time a new sector is read from the SD card this buffer is overwritten and contains the last read content. Subsequent reads of the same SD card sector don't need to be read from the physical card since they are already in the buffer.

Changes to the content of the present sector are always made to this RAM buffer up to the point when the data is committed (SD card write). The sector is generally committed (when it has been modified) when a new sector needs to be read.

This sector buffer avoids unnecessary physical read/writes from/to the SD card, which consume time, and so results in a maximum efficiency when the operation is restricted to a single sector of the card. Since a single sector is often used by directory structures this automatically keeps the amount of writes during directory object manipulations to a minimum.

However, since file operations tend to require interleaved data and file object manipulations, as well as FAT operations, there are often instances where the sector buffer has to be frequently exchanged and so more physical read and write operations performed. In order to be able to better control this behaviour each file that is opened can be defined its own personal operational characteristics and optional data cache.

When a file is opened with the attribute `UTFAT_COMMIT_FILE_ON_CLOSE` it causes changes to the file's object to be avoided until the file is closed. This allows writes and other changes to be made to the data content without the need to maintain the file's object on disk – the file object is maintained locally in the file object only. Avoiding the physical file object update on every file content modification can result in much reduced physical access. Only when the file is closed is the physical file object finally committed. When using the file in this mode it is therefore important to close it after use and it is to be noted that a reset before closing the file will result in new data being lost – it is physically on the disk but the file would still show its old size. *This compromise between speed and reliability needs to be considered in each case*.

When writing file data to the SD card it is most efficient when each write is aligned to either a single sector or else a multiple of sectors; SD card sectors are always 512 bytes in size and aligned on 512 byte boundaries *[writing a new file always with 512 bytes blocks results in optimal performance].* In this case each write is a natural size with optimal efficiency. In some cases however small amounts of data need to be written and or modified, which means that there is generally a requirement to read the physical content, modify and commit again, which can result in a high number of physical read/writes and the inherent delays involved. In such cases a dedicated additional data cache for the file can be of interest. When a file is opened with the attribute `UTFAT_WITH_DATA_CACHE` (only available with option `UTFAT_FILE_CACHE_POOL`) the file is allocated its own sector buffer (when available) and modified data is only committed to the disk when the sector is changed. Reads from the sector (also from other users accessing the file) are taken from the sector buffer (data cache) when its content is up to date. This can generally improve read speed if the reads access cached data and can avoid the need to read/write physical data, especially when numerous small writes are performed.

When multiple users work with the same file, with or without their own data cache, all caches are automatically synchronised so that data remains coherent.

15. Expert Functions

Expert functions are optional capabilities that are activated by enabling the define UTFAT_EXPERT_FUNCTIONS. These functions are especially useful when studying FAT operation or investigating cases of possible data corruption.

In addition to the standard "dir" command a "dird" command allows deleted content to be listed. Since deleted content space may be reused with time deleted files may not always be visible with as use continues.

Eg.

```
D:\>dir
Directory D:\

---A 01.01.2014  12:00             1536 long_file_1.txt
---A 01.01.2014  12:00              768 SHORT_1.TXT
2 files with 2304 bytes
0 directories, 1986248704 bytes free
```

This shows two existing files in a directory.

```
D:\>dird
Directory D:\

[D]---A 01.01.2014  12:00              768 long_file_0.txt
[D]---A 01.01.2014  12:00              256 ~HORT_0.TXT
2 files with 1024 bytes
0 directories, 1986248704 bytes free
```

This shows that there are two deleted file in the directory (that could possibly be recovered). Long file are often visible with their original name whereas short files lost their first letter. Depending on how deleted spaces are reused it is possible deleted files are either no longer visible or their names become corrupted. It is also possible to find multiple deleted files with the same name.

The command "infof" display details about a file or directory.
"infod" displays details about a deleted file or directory.
Eg.

```
D:\>infof short_1.txt
File: short_1.txt is SFN
SNF File located at entry 0x0c in sector 0x00001e01 (cluster 0x00000002)
Data = 0x53 0x48 0x4f 0x52 0x54 0x5f 0x31 0x20 0x54 0x58 0x54 0x20 0x00
0x00 0x00 0x60 0x21 0x44 0x21 0x44 0x00 0x00 0x00 0x60 0x21 0x44 0x03 0x00
0x00 0x03 0x00 0x00
SFN name (archive) = SHORT_1 TXT
File length = 768 starting in sector 0x00001e09 (cluster 0x00000003) FAT
sector 0x0000005f offset 0x03

D:\>infof long_file_1.txt
File: long_file_1.txt is LFN
Starting at entry 0x01 in sector 0x00001e01 (cluster 0x00000002)
First object from 2
```

```
Data = 0x42 0x78 0x00 0x74 0x00 0x00 0x00 0xff 0xff 0xff 0xff 0x0f 0x00
0x4e 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0x00 0x00
0xff 0xff 0xff 0xff
Data = 0x01 0x6c 0x00 0x6f 0x00 0x6e 0x00 0x67 0x00 0x5f 0x00 0x0f 0x00
0x4e 0x66 0x00 0x69 0x00 0x6c 0x00 0x65 0x00 0x5f 0x00 0x31 0x00 0x00 0x00
0x2e 0x00 0x74 0x00
SNF File located at entry 0x03 in sector 0x00001e01 (cluster 0x00000002)
Data = 0x20 0x00 0x0b 0x0f 0x05 0x13 0x12 0x1d 0x2f 0x00 0x00 0x20 0x00
0x00 0x00 0x60 0x21 0x44 0x21 0x44 0x00 0x00 0x00 0x60 0x21 0x44 0x04 0x00
0x00 0x06 0x00 0x00
SFN name (archive) =  ......./.. Alias CS = 0x4e
File length = 1536 starting in sector 0x00001e11 (cluster 0x00000004) FAT
sector 0x0000005f offset 0x04
```

The existing SFN and LFN files are analysed and their details on the disk given. Subsequent use of the "`sect`" command allows the raw data to be viewed at these locations if needed.

The following shows similar examples of the analysis of these two files once they have been deleted.

```
D:\>infod long_file_1.txt
File: long_file_1.txt is LFN
Starting at entry 0x01 in sector 0x00001e01 (cluster 0x00000002)
Deleted LFN
Data = 0xe5 0x78 0x00 0x74 0x00 0x00 0x00 0xff 0xff 0xff 0xff 0x0f 0x00
0x4e 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0x00 0x00
0xff 0xff 0xff 0xff
Deleted LFN
Data = 0xe5 0x6c 0x00 0x6f 0x00 0x6e 0x00 0x67 0x00 0x5f 0x00 0x0f 0x00
0x4e 0x66 0x00 0x69 0x00 0x6c 0x00 0x65 0x00 0x5f 0x00 0x31 0x00 0x00 0x00
0x2e 0x00 0x74 0x00
Deleted LFN
End
SNF File located at entry 0x03 in sector 0x00001e01 (cluster 0x00000002)
Data = 0xe5 0x00 0x0b 0x0f 0x05 0x13 0x12 0x1d 0x2f 0x00 0x00 0x20 0x00
0x00 0x00 0x60 0x21 0x44 0x21 0x44 0x00 0x00 0x00 0x60 0x21 0x44 0x04 0x00
0x00 0x06 0x00 0x00
SFN name (archive) = ......../.. Alias CS = 0x4e
File length = 1536 starting in sector 0x00001e11 (cluster 0x00000004) FAT
sector 0x0000005f offset 0x04

D:\>infod ~HORT_0.TXT
File: ~HORT_0.TXT is SFN
SNF File located at entry 0x0c in sector 0x00001e01 (cluster 0x00000002)
Data = 0xe5 0x48 0x4f 0x52 0x54 0x5f 0x31 0x20 0x54 0x58 0x54 0x20 0x00
0x00 0x00 0x60 0x21 0x44 0x21 0x44 0x00 0x00 0x00 0x60 0x21 0x44 0x03 0x00
0x00 0x03 0x00 0x00
SFN name (archive) = .HORT_1 TXT
File length = 768 starting in sector 0x00001e09 (cluster 0x00000003) FAT
sector 0x0000005f offset 0x03
```

16. exFAT

As the name implies this is an extended implementation of FAT which overcomes some original constraints, such as file size and volume capacity. It modernises the traditional FAT and adds some new conveniences.

This section shows the operation in comparison to FAT32 in order to appreciate what is new and what has been extended, which also helps to understand the overall operation and ideas behind it.

As reference, Windows 10 will tend to default to formatting drives with exFAT that are 32GBytes in size or larger and with FAT32 when smaller, showing that it is preferred for larger volumes; it can however be used for smaller ones from 7MBytes in size.

Disks formatted with exFAT can have the traditional extended boot record as other FAT formatted drives can have. The difference starts in its boot sector which is designed in a way to not be confused with the format of other FAT boot sectors so that other FAT operations won't incorrectly try to interpret it as FAT12, FAT16 or FAT32. The exFAT boot sector is easily recognised as it has the string "EXFAT   " where other FAT boot sectors would have "FAT12   ", "FAT16   " or "FAT32   " as well as various other details that can be checked. Furthermore there is a specific boot sector checksum that can be used to verify that its content is completely as expected to be.

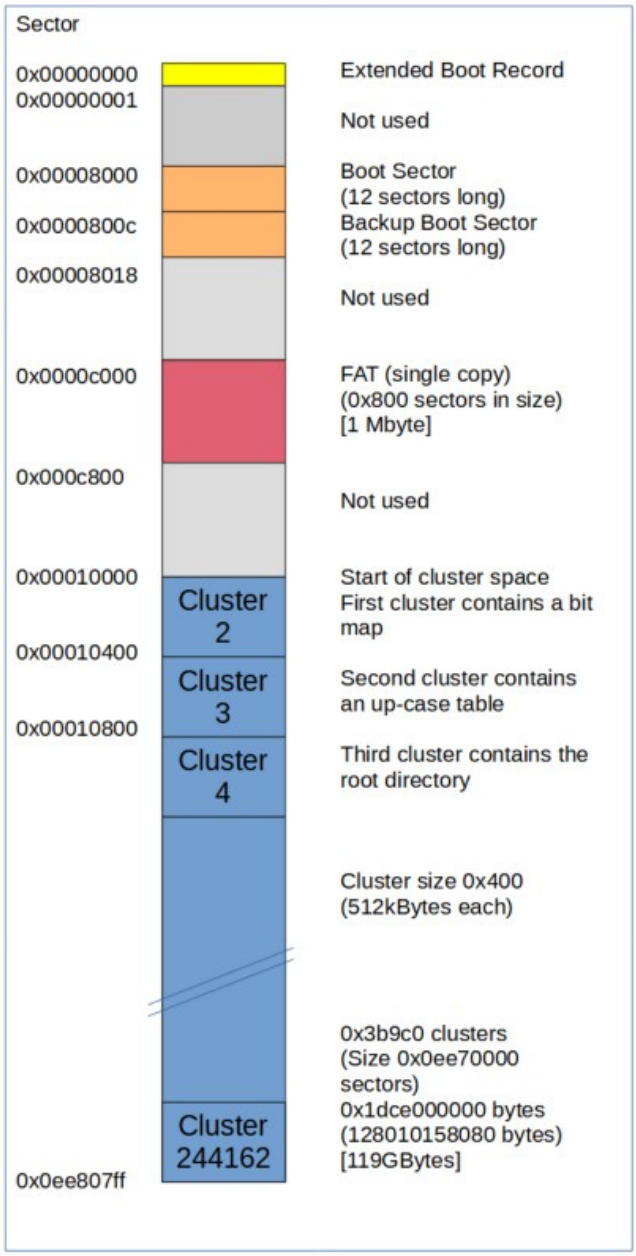
As in the case of the other FAT types there is a copy (backup) of the boot sector that can be used in case the first is found to be corrupted. However the complete boot sector is not a single sector but instead is made up of 12 sectors – the first contains the main information and others can contain extended, or OEM details that are not imperative for operation. The final sector is where the above mentioned boot sector checksum is located. The 32 bit checksum value is simply repeated 128 times in the twelfth sector. The checksum itself is calculated over the complete previous 11 sector content but skips a few fields in the first sector that can change over time. The following is the exFAT boot sector (specifying just the first one as the extended ones are of less interest to this discussion).

```
typedef struct _PACK stBOOT_SECTOR_exFAT
{
    unsigned char BS_jumpBoot[3];       // jump instruction to boot code (0xeb, 0x76, 0x90)
    CHAR          BS_FileSystemName[8]; // string with "EXFAT   "
    unsigned char BS_MustBeZero[53];    // always zero in order to prevent FAT12/16/32
                                           attempting to mount an exFAT volume
    unsigned char BS_PartitionOffset[8];// describes the media-relative sector offset of the
                                           partition which hosts the given exFAT volume
                                           (0 when implementations shall ignore this field)
    unsigned char BS_VolumeLength[8];   // 1MByte .. (2^64 - 1)
    unsigned char BS_FatOffset[4];      // volume-relative sector offset of the first FAT
                                            [(24..ClusterHeapOffset - (FatLength * NumberOfFats)]
    unsigned char BS_FatLength[4];      // length, in sectors, of each FAT table [rounded up
                                           to nearest integer (ClusterCount +
                                           2)*2^2/2^BytesPerSectorShift .. rounded down to
                                           nearest integer (ClusterHeapOffset -
                                           FatOffset)/NumerOfFats]
    unsigned char BS_ClusterHeapOffset[4]; // volume-relative sector offset of the cluster
                                              heap
    unsigned char BS_ClusterCount[4];    // number of clusters the cluster heap contains
    unsigned char BS_FirstClusterOfRootDirectory[4]; // cluster index of the first cluster of
                                                        the root directory.
    unsigned char BS_VolumeSerialNumber[4]; // unique serial number generated by combining the
                                               data and time of formatting the exFAT volume
    unsigned char BS_FileSystemRevision[2]; // exFAT revision number 1.00
    unsigned char BS_VolumeFlags[2];     // flags which indicate the status of various file
                                           system structures on the exFAT volume (not
                                           included in checksum calculation)
    unsigned char BS_BytesPerSectorShift; // 9..12 [512 to 4096 byte sectors]
    unsigned char BS_SectorsPerClusterShift; // 0..(25-BytesPerSectorShift) [1 sector per
                                                cluster..32MB]
```

```
    unsigned char BS_NumberOfFats;        // 1..2
    unsigned char BS_DriveSelect;         // extended INT 13h drive number (often 0x80)
    unsigned char BS_PercentageUse;       // 0..100 - percentage of clusters in the Cluster
                                             Heap which are allocated (not included in
                                             checksum calculation)
    unsigned char BS_Reserved[7];         // reserved
    unsigned char BS_BootCode[390];       // boot-strapping instructions - often set to 0xf4
                                             when not used
    unsigned char ucCheck55;              // this location must be 0x55 - offset 510
    unsigned char ucCheckAA;              // this location must be 0xaa - offset 511
} BOOT_SECTOR_exFAT;
```

As in the FAT32 case the exFAT boot sector contains all information required for the exFAT dimension and storage areas and the resulting sector utilisation can be determined, which looks fairly familiar (compare with Figure 7-1), whereby this is from a 128GByte disk that was formatted by Windows!

Figure 16-1



Beware that the sector map is not to scale since the cluster space is huge and also some unused areas are quite large.

It is not known why there are large unused sectors inserted before the boot sector, Fat and cluster area (other formatters insert only smaller space – according to the specification Fat and cluster heap alignment is foreseen for block alignment to suit the underlying storage medium) but it is seen that an extended boot record is used which indicates the location of the exFAT boot sector(s) at `0x00008000`, which are always present twice.

The exFAT boot sector indicates that there is a single FAT of 1 Mbyte size starting at sector `0x0000c000`. The clusters that it manages are each 512 kbytes in size and occupy 119GBytes starting at sector `0x00010000`.

Unlike FAT32, the formatter already allocates 3 clusters (the first one is numbered as 2 as clusters 0 and 1 never exist – same as FAT32).

New with exFAT is the allocation bitmap in cluster 2 which contains a bit to indicate each cluster that is allocated or reserved. This means that each byte in that cluster can give the present state of 8 clusters, in addition to the information in the FAT itself.

New with exFAT is also the up-case table that has been prepared in cluster 3. This is used for controlling the conversion of lower

case unicode characters to upper case unicode characters that are used in file and directory names.

We notice that the root directory is not in the first cluster but in the one following the bitmap and up-case clusters. The root directory contains a volume entry (as FAT32 does) with an optional name for the volume and will be used to define files and directories in a similar fashion but we will see that the actual entries themselves are not identical – there are also more entry types that can exist!

Up to this point the overall concept remains familiar even if there are some additional details that may later make life easier when dealing with long file and unicode names.

exFAT FAT and Allocation Bitmap

The basic FAT itself is almost the same as the FAT in FAT32; it describes cluster chains in the cluster heap by using (looking at each long word entry in little endien)
`0xfffffff8` – fixed value for the first entry (not used for cluster descriptions)
`0xffffffff` – fixed value for the second entry (not used for cluster descriptions)
For further entries (starting with the first cluster, number 2 up to the end of the cluster heap)
`0x00000000` – free cluster (*but see whether it has been allocated in the bitmap*)
`0x00000002… ClusterCount + 1` – describes the next cluster in a cluster chain
`0xfffffff7` – bad cluster
`0xffffffff` – marks that the corresponding cluster is the final one in a chain

Unlike FAT32, exFAT doesn't necessarily immediately set the first cluster entry to `0xffffffff` when it is used and also doesn't necessarily set it to the next cluster if there is a further contiguous cluster is chained to it. What it does do (which FAT32 doesn't have as a concept) when a cluster is allocated is to mark the cluster as in use in the allocation bitmap and also mark the next as in use if the cluster chain grows.

In the case of the freshly formatted disk the allocation bit map is full of zeros apart from the very first 3 bits which are set to '1', indicating that the clusters 2, 3 and 4 are allocated. Thus, looking at the first long word in the first sector of the allocation bitmap, one finds

`0x00000007` (*first* three available clusters are allocated)

If the root directory cluster (the third cluster – referenced as cluster number 4) were to be grown to include a second contiguous cluster (the fourth cluster – referenced as cluster 5) the allocation of this next cluster would be indicated by changing the bitmap entry to `0x0000000f` (first four available clusters are allocated).

The cluster allocation (whether allocated or free) can thus be seen in the allocation bitmap without needing to look in the FAT – this is why it is not actually necessary to set the FAT cluster entry values at all as long as the cluster chain is contiguous. *Free cluster counting is faster by reading the allocation bitmap and counting the number of bits that are zero, than needing to read the complete FAT itself.*

As soon as cluster chains are not contiguous the FAT entries do need to be set – eg. `0x00000000 0x00000000` of a contiguous allocated chain, will need to be marked accordingly (eg. `0x00000005 0xffffffff`) when subsequent ones are allocated to other chains.

This shows that anyone with FAT32 understanding should have no real difficulties understanding exFATs cluster allocation and management since it is very similar to FAT32.

Directory Entries

FAT32 uses directory entries to define directories and files and can use multiple such when working with long file names. These are 32 bytes in size.

exFAT uses a similar technique, with 32 byte entries, but extends with a number of such types which are used for different purposes and also types that can be extended for vendor use.

FAT32 volumes can have an optional (up to 11 ASCII characters long) volume name, which is stored in its boot sector. exFAT instead us its extended directory entry capability to move this into the root directory and allows it to be up to 11 unicode characters long. The new volume label entry is defined as

```
typedef struct _PACK stEXFAT_VOLUME_LABEL_DIRECTORY_ENTRY
{
    unsigned char EntryType;         // EXFAT_VOLUME_LABEL_IDENTIFIER 0x83
    unsigned char CharacterCount;    // length of the Unicode string the VolumeLabel field
                                        contains (max. 22 bytes)
    unsigned char VolumeLabel[22];   // user-friendly name of the volume (unicode - 11
                                        characters/22 bytes)
    unsigned char Reserved[8];       // reserved
} EXFAT_VOLUME_LABEL_DIRECTORY_ENTRY;
```

This, and all other such exFAT directory entries, are 32 bytes in size and the first byte specifies their type.

Whereas FAT32 uses a single directory entry for a file or directory that have a short filename exFAT always uses at least three entries. To describe a file or directory a group of
- file directory identifier entry [file attributes, time stamps, etc.]
- stream identifier entry [name length, file size, content location, etc.]
- file name identifier entry [up to 15 unicode characters of the file name]
are used

When names greater than 15 unicode characters are used multiple file name identifier entries are used (similar to FAT32 LFN occupation of multiple directory entries).

These three directory entry types are described by

```
typedef struct _PACK stEXFAT_FILE_DIRECTORY_ENTRY
{
    unsigned char EntryType;            // EXFAT_FILE_DIR_IDENTIFIER 0x85
    unsigned char SecondaryCount;       // describes the number of secondary directory entries
                                           which immediately follow the given primary
                                           directory entry
    unsigned char SetChecksum[2];
    unsigned char FileAttributes[2];
    unsigned char Reserved1[2];
    unsigned char CreateTimeStamp[4];
    unsigned char LastModifiedTimestamp[4];
    unsigned char LastAccessedTimestamp[4];
    unsigned char Create10msIncrement;
    unsigned char LastModification10msIncrement;
    unsigned char CreateUtcOffset;
    unsigned char LastModifiedUtcOffset;
    unsigned char LastAccessedUtcOffset;
    unsigned char Reserved2[7];
} EXFAT_FILE_DIRECTORY_ENTRY;
```

```
typedef struct _PACK stEXFAT_STREAM_EXTENSION_DIRECTORY_ENTRY
{
    unsigned char EntryType;             // EXFAT_STREAM_IDENTIFIER 0xc0
    unsigned char GeneralSecondaryFlags; //
    unsigned char Reserved1;             //
    unsigned char NameLength;            // 1..255
    unsigned char NameHash[2];           // 2-byte hash of the up-cased file name
    unsigned char Reserved2[2];          //
    unsigned char ValidDataLength[8];    //
    unsigned char Reserved3[4];          //
    unsigned char FirstCluster[4];       // first cluster of an allocation in the cluster heap
                                            associated with the given directory entry
    unsigned char DataLength[8];         // the size, in bytes, of the data the associated
                                            cluster allocation contains
} EXFAT_STREAM_EXTENSION_DIRECTORY_ENTRY;
```

```
typedef struct _PACK stEXFAT_FILE_NAME_DIRECTORY_ENTRY
{
    unsigned char EntryType;             // EXFAT_FILE_NAME_IDENTIFIER 0xc1
    unsigned char GeneralSecondaryFlags; //
    unsigned char FileName[30];          // a unicode string, which is a portion of the file
                                            name (unused characters set to 0x0000)
} EXFAT_FILE_NAME_DIRECTORY_ENTRY;
```

Various others are described in the exFAT specification but not detailed here as they are not commonly needed.

In order to find a file and its content (in the cluster heap) a directory is scanned for these directory groups until the name is matched and the information in the (at least 3 contiguous) directory entries used. This is in fact very similar to finding a LFN FAT32 file in a directory and so is not any more complicated to do.

A quick comparison with FAT32 directory entries reveals a few interesting extensions:
- a 16 bit checksum
- inherent support of unicode file names of up to 255 unicode characters in length
- data length of up to `0xffffffffffffffff`, which is a huge 128PB
- a file name hash, which allows faster file name matching when long file names are used

Long File Names and Up-case Table

As noted in the previous section file names of up to 255 unicode characters are supported by exFAT. In a similar way to FAT32 LFN, longer file names are added using further directory entries as needed, whereby any name with up to 15 unicode characters will fit in a single file name directory entry. For each file or directory name an additional two (file entry and stream extension entry) are always needed too.

Possibly the most interesting extension is the 16-bit `NameHash` in the stream extension entry. This is added when the file or directory is created by hashing the up-cased version of the file/directory name: The mechanics involved is to first make an up-cased copy of the file/directory name [for ASCII characters that is very easy since it just means using the

capital version of each character but for unicode it involves the use of the up-case table, as will be described later] after which the up-cased string is used as input to the hash algorithm. The two byte result is then inserted at this location.

When a file name is searched, the same process can be used in order to generate its name hash, which can then be used to match with the hash stored in each stream extension entry. Once a hash match has been found it means that it is 'highly probable', but not guaranteed, that the searched file entry has been found. To ensure the match a final complete byte for byte match of the file name against the searched name is performed. This is certainly more efficient when searching for a file or directory with a (very) long file name in a directory with many entries but can in fact be skipped and the name simply be matched – *in fact this is often more efficient when searching in directories with only a few entries*.

17. Conclusion

The operation of SD-cards and FAT has been described in enough detail for users to understand the internal workings of **µtFAT V2.1.** µtFAT can be configured to work with various features allowing basic operation in simple systems neither requiring long file name (LFN) support nor write operation (smallest code requirements) through systems which necessitate full long file name capabilities, optimised performance and expert functions to monitor and analyse the SD card / memory stick FAT contents.

The user interface to **µtFAT V2.1**, allowing applications to efficiently utilise the operations in a simple and logical manner, has been described with use examples.

V0.08 adds a complete list of user interface commands and reworked description of FAT32 operation, plus notes about SDIO interface support.

V0.09 adds LFN details.

V0.10 adds LFN rename and creation.

V2.00 adds utTruncateFile(), LFN rename details, data caching and description of expert functions

V2.10 adds FAT12 and exFAT details

18. Disclaimers

The information contained in this document is presented only as an overview of SD cards and FAT and is provided "AS-IS" without any representations or warranties of any kind. No responsibility is assumed by the µTasker developers for any damages or infringements of patents which may result from its use. No license is granted by the µTasker developers implication, estoppel or otherwise under any patent or other rights of the µTasker developers or any third party. Nothing herein shall be construed as an obligation by the µTasker developers to disclose or distribute any technical information, know-how or other confidential information to any third party.

The µtFAT module is offered “AS-IS” for users of the µTasker project for hobby and/or commercial work. In the case of commercial applications a basic µTasker commercial license for the used processor is implied. The module is supported by the µTasker developers and all attempts will be made to correct any operation which proves to be faulty but the licensee/user accepts that no claims for compensation may be made, for any reasons whatsoever, whether due to µTasker code, its environment and tools or use thereof.

To this effect please ensure that development work is not performed with SD cards/memory sticks containing important data – potential data loss can be simply avoided by using SD cards/memory sticks reserved exclusively for experimental and development work; please adhere to this simple rule and have fun with the µtFAT module!