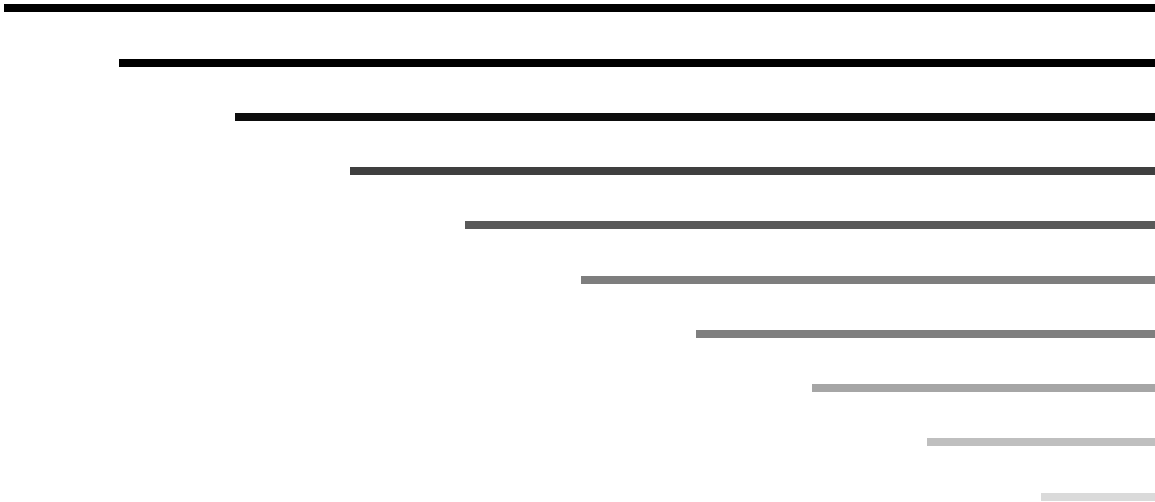


*Embedding it better...*



μTasker Document

**μTasker – utFAT**



## Table of Contents

1.	Introduction .....	3
2.	Electrical Connection to the SD card in SPI Mode .....	5
3.	SPI Mode of Operation .....	7
4.	SD Card Initialisation Phase .....	8
5.	SD Card Disk Mounting .....	9
6.	First Steps with an SD Card and Understanding FAT32.....	12
6.1.	Checking the Details of a Non-Formatted SD Card .....	13
6.2.	Formatting or Re-formatting an SD Card .....	14
6.3.	Displaying the Content of a Freshly Formatted SD Card .....	17
6.4.	Creating a New Directory.....	21
6.5.	Creating a New File.....	22
7.	FTP Server and µtFAT .....	25
8.	HTTP Server and µtFAT .....	25
9.	Working with the µtFAT User Interface.....	26
10.	µtFAT User Interface.....	28
10.1.	utAllocateDirectory() .....	28
10.2.	utOpenDirectory().....	29
10.3.	utOpenFile().....	30
10.4.	utSeek().....	31
10.5.	utWriteFile() .....	32
10.6.	utReadFile() .....	33
11.	µtFAT File Management.....	34
12.	Long File Name Support.....	34
13.	Conclusion.....	34
14.	Disclaimers.....	36

## 1. Introduction

**μtFAT V1.0** is a FAT32 compatible file system for use with V2 SD cards [*Secure Digital*] and HCSD cards [*Secure Digital High Capacity*]. It uses the SPI mode of these 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 and the total storage space from typically (at the time of writing) 2 GByte to 16 GByte, but up to 2TByte possible (with 512 byte sector size).

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 demo project for simple test and study. Furthermore μtFAT and SD cards 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 long file names (*LFN*). It can optionally format disks, create, rename and delete directories and files, plus write file content. μtFAT V1.0 doesn't support creating new directories and files with long file names; the module is however prepared to include optionally generating long file names in future extensions. [*Note that the fact that no LFNs are created should exclude the use of this version of μtFAT from any possible patent infringements*]

μtFAT neither supports FAT12 nor FAT16, nor V1 SD cards nor MMCs. Due to the fact that it is becoming increasingly difficult to purchase SD cards rather than HCSD cards the smaller, older ones are considered legacy devices - μtFAT V1.0 arrives at the time when it makes 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.

Since SD cards 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 is not detected the internal file system falls 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
- FAT32 with optional long file name (LFN) support
- Memory management interface controlling automated detection and mounting of the SD card and enabling supervision of file protection and sharing
- Simplified user interface incorporating a variety of standard tasks like listing directories
- Full integration into μTasker HTTP and FTP servers with fall-back capability to μFileSystem in internal FLASH or external SPI FLASH

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 FAT32 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 FAT32 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 FAT32 operation, especially when working together with the μTasker simulator and its SD card simulation capabilities
- To discuss the relevant details about file management (sharing and protection) in a multi-user embedded system with real-time demands

The following references serve as basis for SD card and FAT32 specifications:

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

There are many additional sources for information available in the Internet so this document restricts details to that of 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 and UC3B
- Freescale M522XX
- Luminary Micro LM3Sxxxx
- NXP LPC2XXX
- ST STR91XF

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, copying data to the card via FTP, serving web server content up to the capacity of the SD card as well as dynamic content generation, fall back to the μFileSystem when no SD card is inserted as well as the complete functionality of the μTasker V1.4 project.*

**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.

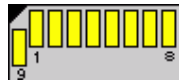


Figure 2-1 Standard SD card pin numbering

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

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 most processors have 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 HCSD 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 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 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 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.
- 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 no 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 the most significant bit of the result value (0x80). 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 expected to be the IDLE stat (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.

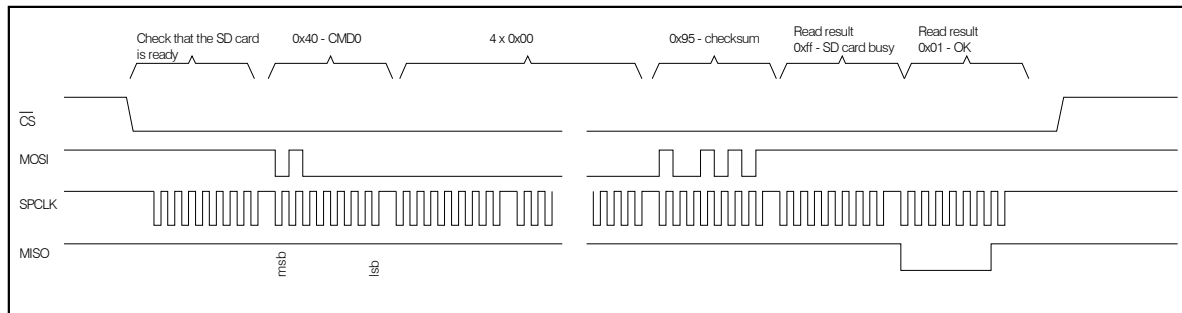


Figure 3-1 Example of SPI mode command – CMD0

## 4. SD Card Initialisation Phase

The initialization of the SD card is managed by the mass-storage task. This task is activated in the demo 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 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 300kHz to the higher operating speed of up to 25MHz.

## 5. 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 FAT32 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 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 always 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 (HCSD 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.

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, 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 a PC) or by commanding the formatting if the utFAT 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 5-1 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 5-2 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 5-3 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
    // typically 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_FATsZl6[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 5-4 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 V1.0 will use just 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.

After reading this sector, 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 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 so is not supported. 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 valid mounting is complete. From this point on the SD card can be used.
- 3) To find out important details about the SD card and its FAT32 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 V1.0 includes an SD card simulator with a 2G SD-card simulated as default. 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.

## 6. 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 not FAT32. In this case it needs to be reformatted
- *FAT32 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)
- *FAT32 formatted with data* – in this case it contains directories and/or files and probably 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!

## 6.1. 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 utFAT 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:

```

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     [label] format (unformatted) disk
re-format  [label] reformat disk!!!!
sect       [hex number] display sector
help       Display menu specific help
quit       Leave command mode

```

Terminal 6-1 Disk Interface Menu

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

```

SD-card not formatted (1967128576 bytes)
CSD: 0x00 0x2e 0x00 0x32 0x5b 0x5a 0x83 0xa9 0xff 0xff 0xff 0x80 0x16 0x80 0x00 0x55

```

Terminal 6-2 “info” command response from an unformatted SD card

This is showing that the SD card is not formatted but has about 2 GByte 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.

## 6.2. 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 `utFAT` 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!!**

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 take place by writing the partition information and boot sector information accordingly as well as resetting all FAT32 content. In addition a volume label can be passed with the formatting command (up to 11 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 (1967128576 bytes)
Bytes per sector: 512
Cluster size: 4096
Directory base: 0x00000002
FAT start: 0x0000005f
FAT size: 0x00000ea5
Number of FATs: 2
LBA: 0x00001da9
Total clusters: 0x0007504a
Info sect: 0x00000040
Free clusters: 0x00075249
Next free: 0x00000003
CSD: 0x00 0x2e 0x00 0x32 0x5b 0x5a 0x83 0xa9 0xff 0xff 0xff 0x80 0x16 0x80 0x00 0x55
```

Terminal 6-3 “info” command response from a formatted SD card

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.

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 utFAT 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 0x00000000
0x380b000c 0x0003fb8f8 0x9fc10000 0x0000003a 0x00000000 0x00000000 0x00000000 0x00000000
0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0xaa550000
```

Terminal 6-4 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 little-endian – 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 generally chosen.*

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.



### 6.3. 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, 1963233280 bytes free
D:\>
```

Terminal 6-6 “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 copied 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 6.1.

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 0xea5 sectors in size. It has in addition informed us that there are two FAT32s (they are copies in case one were to 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.

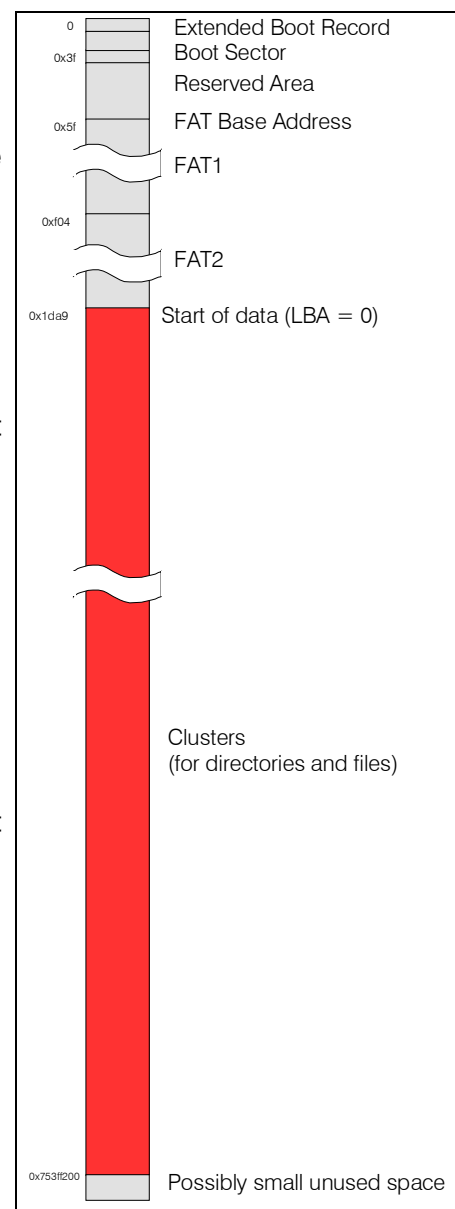


Figure 6-1 SD card sector utilisation

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 can be used by FAT32 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 0xea5. Other combinations may also be possible – for example larger cluster sizes, requiring less FAT32 space, or smaller cluster sizes requiring larger FAT32 space. 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 space that can be allocated to any object (file or directory) so any object will be constructed of either 1 cluster or a multiple of clusters. When a file is created with zero 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.

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 situated. In fact it requires one long word of FAT32 space to manage one cluster, 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 versa. If the cluster size is chosen too small 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 0xea5 sectors in size (where a sector is 512 bytes).
- Therefore the FAT32 area is  $0xea5 \times 0x200$  (or  $3749 \times 512$ ) =  $0x1d4a00$  (1'919'488) bytes in size.
- One long word (32 bits) in the FAT32 manages a single cluster (this is explained more below), meaning that this FAT32 can manage  $0x1d4a00 / 4 = 0x75'280$  (479'872) clusters.
- The cluster size is 4k so the cluster area that can be managed is  $0x75'280 \times 0x1000 = 0x7828000$  (1'965'555'712) bytes in size. *It will be seen that usually 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 1'967'128'576 bytes available for use and the reserved area plus the FAT32s use up 0x1da9 sectors (0x3b520 or 3'887'616*

bytes), leaving 0x7504ae00 (19'632'40'960) bytes for the cluster area. This makes 0x7404a (exactly 479306.875) clusters in total for actual use. The 0.875 clusters (3'584 bytes) are the 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 0x0fffffff7 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 content of the FAT32 area, so here it is:

```
D:\>sect 5f
Reading sector 0x0000005f
0xffffffff 0xffffffff 0xffffffff 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 6-6 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 are free.

The first two long word values are standard entries which are described in the Microsoft *FAT32 File System Specification* but otherwise of no significance to us. They are never used but need to be there. The third long word 0xffffffff is an 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 0xffffffff means that the cluster is 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 not 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 (LBS) which starts at sector 0x1da9 (see “info” command). Don’t forget that it does in fact occupy at least one cluster space so its content is not sector 0x1da9 alone but also the sectors up to and including 0x1db0.

```
D:\>sect 1da9

Reading sector 0x00001da9
0x41465455 0x00000054 0x08000000 0x00000000 0x00000000 0x00000000 0x00000000 0x0000...
0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x0000...
0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x0000...
0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x0000...
0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x0000...
0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x0000...
0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x0000...
0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x0000...
D:\>
```

Terminal 6-7 Content of empty root directory

This sector’s content is now from 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 can 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 6-1 Directory entry struct

Analysing the single directory entry in the root directory reveals the volume name “UTFAT”, with attribute “volume ID”. There are no further entries because the following directory entry starts with a 0x00. Note further that there are no time and date stamps used by utFAT V1.0.

## 6.4. Creating a New Directory

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:\
---- 03.01.2010  14:55 <DIR>          dir1
0 files with 0 bytes
1 directories, 1965539328 bytes free
D:\>
```

Terminal 6-8 Creating and listing a new directory

utFAT adds a fixed time and date stamp when directories and files are created and when they are written. This can be replaced by a routine to set the information from a RTC (Real Time Clock) or similar if an accurate time stamp is required.

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

```
D:\>sect 1da9
Reading sector 0x00001da9
0x41465455 0x00000054 0x08000000 0x00000000 0x00000000 0x00000000 0x00000000 0x000...
0x31524944 0x20202020 0x10202020 0x00000018 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 6-9 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.

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

```
D:\>sect 5f
Reading sector 0x0000005f
0x0fffffff 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 6-10 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  $0x1da9 + 8 = 0x1db1$  (remembering that each cluster is made up of 8 sectors).

```
D:\>sect 1db1

Reading sector 0x00001da9
0x2020202e 0x20202020 0x10202020 0x00000000 0x00000000 0x00000000 0x00020000 0x00000000
0x2020202e 0x20202020 0x10202020 0x00000000 0x00000000 0x00000000 0x00020000 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...
D:\>
```

Terminal 6-11 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 µFAT module generates these for compatibility but doesn’t actually use them.

## 6.5. Creating a New File

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 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” [ensure that long file names are not used since µFAT only supports short file names and will not accept it otherwise].
- 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 hoped 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
0x0fffffff8 0xfffffffff 0xfffffffff 0xfffffffff 0x00000005 0xfffffffff 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 6-12 Content of first FAT32 sector – including one file occupying multiple clusters

This shows the result of a file of 6651 bytes in length. It was originally created in cluster 4 (it started out with `0xffffffff`) but then overflowed and required a new cluster. Since the next cluster was free it then started occupying cluster 5, which has the value `0xffffffff` 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.

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:

```
D:\>sect 1db1

Reading sector 0x00001da9
0x2020202e 0x20202020 0x10202020 0x00000000 0x00000000 0x00000000 0x00020000 0x00000000
0x2020202e 0x20202020 0x10202020 0x00000000 0x00000000 0x00000000 0x00020000 0x00000000
0x45444e49 0x20202058 0x204d5448 0x00000018 0x00000000 0x00000000 0x00040000 0x000019fb
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 6-13 Content of the empty directory "dir1" with new file "index.htm"

The file is recognised by its file attribute `0x20`, has a length of `0x19fb` 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 it 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 by seeing whether this is the final cluster (a value of `0xffffffff` would be expected) or whether there are following clusters. Since this present cluster is 4 it is corresponding FAT32 entry can be found by reading the first FAT32 sector and then using the 4<sup>th</sup> cluster entry (see Terminal 6.12) to see which cluster to use next. In this case, simply use the following cluster number 5.

Having the sector display as a 16 x 8 field makes following FAT32 cluster chains simple since the present cluster can be read directly – eg. if the present cluster is 0x13, its corresponding entry is one line down and 4 entries to the right. If this is 0x0ffffff it is the final cluster in the chain, else the next one is directly displayed.

Generally the calculation for a cluster is as follows:

- 1) The FAT32 sector in which the cluster entry is to be found is  $(\text{cluster}/128)$ . The actual physical sector is  $\text{FAT\_start} + (\text{cluster}/128)$
- 2) The entry in this sector is  $(\text{cluster} \& 0x7f)$

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 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 has, in comparison, no benefits.

## 7. FTP Server and µtFAT

The FTP server can work with the SD card if the define `FTP_UTFAT` is active. If a formatted SD card 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.

Since µtFAT V1.0 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. It can only create directories and new files with short file names (8.3 format).

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 relative to the present directory position.

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

## 8. HTTP Server and µtFAT

The HTTP server can work with the SD card if the define `HTTP_UTFAT` is active. If a formatted SD card 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 V1.0 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 relative to the HTTP root.

If the SD card 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 first release doesn't support posting data to the SD card – this is however not a restriction of the µtFAT module but the presently missing integration in the HTTP server.*

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, 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 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 dynamic HTTP methods can be applied to such HTTP content if required.

## 9. Working with the µFAT User Interface

The µFAT user interface shows how moving around the directory contents can be performed in a simple manner similar to the well known DOS interface. Terminal 9.1 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 9-1 Content of an SD card containing directories and files using LFN

As shown in Terminal 9-2 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 cellpadding=0 cellspacing=0><tr><td width=10%>&nbsp;</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 9-2 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 commands are (commands are case-sensitive):

info	utFAT/card info	Display SD card and FAT32 information
dir	[path] show directory content	Display present directory, eg. “ <code>dir dir1/webpages</code> ”
cd	[path] change dir. (.. for up)	Move the directory location, eg. “ <code>cd dir/webpages</code> ”
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 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. “ <code>mkdir dir2</code> ”. If the directory already exists it will not be modified.
rename	[from] [to] rename	Rename an existing file or directory, eg. “ <code>rename dir1 dir2</code> ”
print	[path] print file content	Print the content of an existing file to the debug output (non-displayable characters are output as ‘.’)
del	[path] delete file or dir.	Delete a file or directory. Directories can only be deleted when they are empty.
format	[label] format (unformatted) disk	Format an unformatted SD card with optional volume ID, eg. “ <code>format UTFAT_DISK</code> ”
re-format	[label] reformat disk!!!!	Reformat a formatted disk, with optional volume ID. <b>WARNING – the existing FAT32 table will be destroyed!!</b>
sect	[hex number] display sector	Display the content of a 512 bytes sector on the SD card, eg. “ <code>sect 5f</code> ”

## 10. µtFAT User Interface

### 10.1. 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`.

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).

```
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) {
    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 and FTP don't have a path string but instead all is referenced to their individual root directories – FTP can still move down to sub-directories but not back up in directory steps – just back to its root directory).

\* 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 interfaces uses a path string to accomplish this.

FTP doesn't use a path string. It can still reference directives below itself

"dir dir2/dir3/dir4" and it can also move down "cd dir2/dir3" but it cannot move back up the path nor reference upwards like "cd ../dir5". In this case cd .. simply moves up to its original root directory.

Therefore the string use is user definable. FTP and 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.

## 10.2. 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 objects 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 call

```
if (ptr_utDirectory->ucDirectoryFlags & UTDIR_VALID)
```

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.

### 10.3. utOpenFile()

```
extern int utOpenFile(const CHAR *ptrFilePath, UTFILE *ptr_utFile, unsigned short usAccessMode);
```

This function opens a file referenced by `ptrFilePath` 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\_FILE\_IS\_DIR - the file is a directory type

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

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\_SEARCH\_INVALID - a file search was invalid since the file object is not associated with a directory object

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 and the file object's `ptr_utDirObject` must be set to the directory pointer.

```
UTFILE utFile; // temporary file object
utFile.ptr_utDirObject = ptr_utDirectory;
if (utOpenFile(ptrInput, &utFile, (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 open 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 open 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)
```

#### 10.4. 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 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 `lseek()` library function.

## 10.5. 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

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. This allows content to be overwritten. If additional content is to be added the file position should first be change accordingly using the `utSeek()` command. 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 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.

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 ", 13) != 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.*

## 10.6. 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 written.

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 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 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.*

```
extern int utMakeDirectory(const CHAR *ptrDirPath, UTDIRECTORY *ptrDirObject);
extern int utLocateDirectory(const CHAR *ptrDirPath, UTLISTDIRECTORY *ptrListDirectory);
extern int utChangeDirectory(const CHAR *ptrDirPath, UTDIRECTORY *ptrDirObject);
extern int utDeleteFile(const CHAR *ptrFilePath, UTDIRECTORY *ptrDirObject);
extern int utListDir(UTLISTDIRECTORY *ptr_utDirectory, FILE_LISTING *ptrFileLists);

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

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

extern int fnReadSector(unsigned char ucDisk, unsigned char *ptrBuffer, unsigned long
ulSectorNumber);
extern int utFreeClusters(unsigned char ucDisk, UTASK_TASK owner_task);
extern UTDISK *fnGetDiskInfo(unsigned char ucDisk);
```

These commands are used by the disk interface, which serves as additional reference.

### To do...

In the meantime please use the µFAT forum for questions and answers:  
<http://www.utasker.com/forum/index.php?board=10.0>

## 11. µFAT File Management

### To do...

In the meantime please use the µFAT forum for questions and answers:  
<http://www.utasker.com/forum/index.php?board=10.0>

## 12. Long File Name Support

### To do...

In the meantime please use the µFAT forum for questions and answers:  
<http://www.utasker.com/forum/index.php?board=10.0>

## 13. Conclusion

In work. Not officially released.



## 14. Disclaimers

*The information contained in this document is presented only as an overview of SD cards and FAT32 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 containing important data – potential data loss can be simply avoided by using SD cards reserved exclusively for experimental and development work; please adhere to this simple rule and have fun with the μtFAT module!*