AN10916

FAT library EFSL and FatFs port on NXP LPC1700

Rev. 2 — 6 July 2010 Application note

Document information

Info	Content
Keywords	LPC1700, File system, EFSL, FatFs, SDC/MMC
Abstract	EFSL and FatFs are two popular FAT libraries for developing small embedded systems. This application note describes how to port these FAT libraries to NXP Cortex-M3 LPC1700 devices. An external SDC/MMC, connected to an LPC1700 SPI/SSP0 will be used as a physical disk. A set of easy-to-use SPI and SDC/MMC API functions are also provided.

Revision history

Rev	Date	Description
2	20100706	Added text “and applicable licenses and/or copyrights” to sentence regarding URLs for FAT, EFSL, and FatFs.
1	20100304	Initial version.

Contact information

For additional information, please visit: http://www.nxp.com

For sales office addresses, please send an email to: salesaddresses@nxp.com

1. Introduction

EFSL and FatFs are two popular FAT libraries for developing small embedded systems. This application note describes how to port these two FAT libraries to NXP Cortex-M3 LPC1700 devices.

A set of easy-to-use SPI and SDC/MMC API functions is also provided to access SDC/MMC conveniently.

This application note includes:

- A set of easy-to-use SPI and SDC/MMC APIs to access the SDC/MMC via SPI on LPC1700
- How to port EFSL and FatFs to LPC1700 step by step

The sample software is tested on Keil's MCB1700 evaluation board with a 2 GB Kingston Micro SD card.

2. Access SDC/MMC via SPI on LPC1700

2.1 SDC/MMC introduction

The **Secure Digital Card** (SDC below)/ **Multi Media Card** (MMC below) is a flash-based memory card specifically designed to meet the security, capacity, performance and environmental requirements inherent in next generation mobile phones and consumer electronic devices.

SDC communication is based on an advanced nine-pin interface (clock, command, 4xData and 3xPower lines) designed to operate in a low voltage range. The SDC host interface supports regular MMC operation as well. There are also reduced size versions, such as RS-MMC, miniSD and microSD, with same function.

The main difference between the SDC and MMC is the initialization process.

2.2 SDC/MMC interface

The SDC/MMC interface allows for easy integration into any design, regardless of microcontroller used. For compatibility with existing controllers, the SDC/MMC offers, in addition to the SDC/MMC Interface, an alternate communication protocol based on the SPI standard.

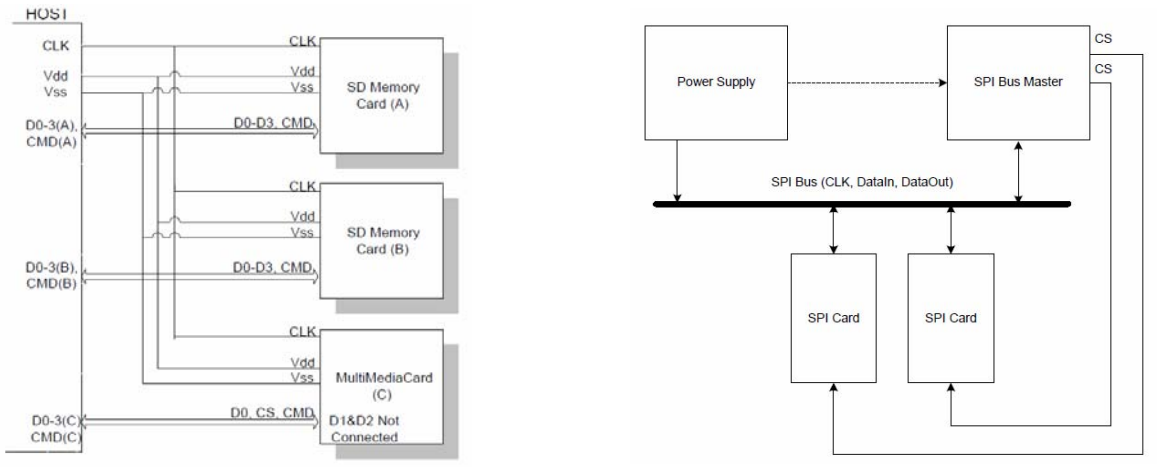
The SDC/MMC pin assignment is shown in Table 1:

Table 1. SDC/MMC pin assignment

Pin No.	Name	Type	Description
SDC/MMC Bus Mode[1]			
1	CD/DAT3	I/O, PP	Card detect/Data line [Bit 3]
2	CMD	I/O, PP	Command/Response
3	Vss1	S	Supply voltage ground
4	Vdd	S	Supply voltage
5	CLK	I	Clock
6	Vss2	S	Supply voltage ground
7	DAT0	I/O, PP	Data line [Bit 0]
8	DAT1	I/O, PP	Data line [Bit 1]
9	DAT2	I/O, PP	Data line [Bit 2]
SPI Bus Mode			
1	CS	I	Chip Select (active low)
2	DataIn	I	Host-to-card Commands and Data
3	Vss1	S	Supply voltage ground
4	Vdd	S	Supply voltage
5	CLK	I	Clock
6	Vss2	S	Supply voltage ground
7	DataOut	O	Card-to-host Data and Status
8	RSV	---	Reserved
9	RSV	---	Reserved

[1] For MMC, only one data line, DAT0, is used.

The SDC/MMC bus topology on both modes is shown in Fig 1.



a. SDC/MMC bus mode

b. SPI bus mode

Fig 1. SDC/MMC system bus topology

Since LPC1700 does not have SDC/MMC native host interface, we have to access the SDC/MMC via SPI interface. Only SPI mode will be discussed in the rest of this section.

2.3 SPI mode

2.3.1 SPI bus topology

The SPI mode is a secondary communication protocol for SDC/MMC. This mode is a subset of the SDC/MMC protocol, designed to communicate with an SPI channel, commonly found in NXP and other vendors' microcontrollers.

The SDC/MMC can be attached to most microcontrollers via the generic SPI interface. Because SPI mode is suitable for low cost embedded applications, there is no reason to attempt to use native mode with a cheap microcontroller that has no native SDC/MMC interface.

The SDC/MMC identification and addressing algorithms are replaced by the hardware CS signal. A card (slave) is selected for every command by asserting the CS signal (active low). Refer to Fig 1(b).

The CS signal must be continuously active for the duration of the SPI transaction (command, response and data).

The bi-directional CMD and DAT lines are replaced by unidirectional dataIn and dataOut signals. This eliminates the ability to execute commands while data is being read or written which prevents sequential multi read/write operations.

2.3.2 SPI bus protocol

The SPI standard defines the physical link only and not the complete data transfer protocol. In SPI mode, the SDC/MMC uses a subset of the SDC/MMC protocol and command set.

Similar to the SDC/MMC bus protocol, the SPI messages are built from command, response and data-block tokens. The host (master) controls all communication between host and cards. The host starts every bus transaction by asserting the CS signal, low.

The response behavior in SPI mode differs from the SDC/MMC mode in the following three ways:

1. The selected card always responds to the command.
2. An 8-bit or 16-bit response structure is used.
3. When the card encounters a data retrieval problem, it will respond with an error response (which replaces the expected data block) rather than time-out as in the SDC/MMC bus mode.

2.3.3 Mode selection

The SDC/MMC wakes up in the SDC/MMC mode. It will enter SPI mode if the CS signal is asserted (negative) during the reception of the reset command (CMD0). If the card recognizes that the SDC/MMC mode is required it will not respond to the command and remain in the SDC/MMC mode. If SPI mode is required, the card will switch to SPI mode.

2.4 SPI interface on LPC1700

There is one SPI controller with synchronous, serial, full duplex communication and programmable data length on LPC1700 devices.

Remark: SSP0 is intended to be used as an alternative for the SPI interface, which is included as a legacy peripheral. Only one of these peripherals can be used at any one time.

The SSP can produce a faster data bit rate than SPI. The maximum SPI data bit rate is one eighth of the input clock rate, and the maximum SSP speed (in master mode) is pclk/2. In Slave mode, the SSP clock rate provided by the master must not exceed 1/12 of the SSP peripheral clock (selected in peripheral clock selection register).

For example, if the PCLK is set to 100 MHz, the maximum SPI rate will be 12.5 Mbit/sec (100 MHz/8). The maximum SSP speed in master mode will be 50 Mbit/sec (100 MHz/2) and in slave mode 8Mbit/sec (100 MHz/12).

The pin connection between LPC1700 SSP0 and external Micro SD card slot on Keil's MCB1700 board is shown in Fig 2.

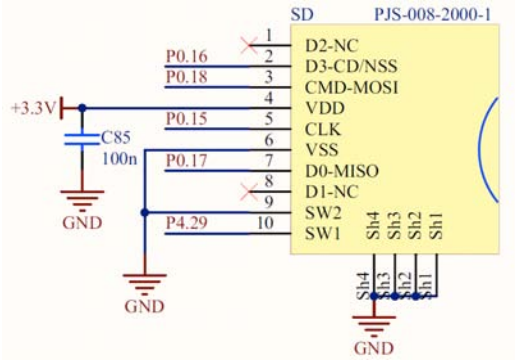


Fig 2. Schematic of SDC/MMC on Keil MCB1700 board

Table 2. Micro SD card pin assignment

Note: The pin assignment of an SD card and Micro SD card is slightly different

Pin No.	Name	Type	Description
SDC Bus Mode			
1	DAT2	I/O, PP	Data line [Bit 2]
2	CD/DAT3	I/O, PP	Card detect/Data line [Bit 3]
3	CMD	I/O, PP	Command/Response
4	Vdd	S	Supply voltage (2.7v / 3.6v)
5	CLK	I	Clock
6	Vss	S	Supply voltage ground
7	DAT0	I/O, PP	Data line [Bit 0]
8	DAT1	I/O, PP	Data line [Bit 1]
SPI Bus Mode			
1	RSV	---	Reserved
2	CS	I	Chip Select (active low)
3	DataIn	I	Host-to-card Commands and Data
4	Vdd	S	Supply voltage
5	CLK	I	Clock
6	Vss	S	Supply voltage ground
7	DataOut	O	Card-to-host Data and Status
8	RSV	---	Reserved

2.5 SPI drivers and APIs

A total of eight API functions are provided for the SPI communication in lpc17xx_spi.c:

- void LPC17xx_SPI_Init (void);

  This API is used to initialize SPI interface on LPC1700 through configuring SSP0 PCONP, GPIO, control registers.

- void LPC17xx_SPI_DeInit (void) ;

  This API is used to clear the initial SSP0 registers' configurations and then power off the SSP0.

- void LPC17xx_SPI_Select (void);

  This API is used to assert the CS (low).

  The host starts every bus transaction by asserting the CS signal low, and the CS signal must be asserted during a transaction. If the CS signal goes high any time during a data transfer, the transfer is considered to be aborted. This signal is not directly driven by the master. It could be driven by a simple general purpose I/O under software control.

- void LPC17xx_SPI_DeSelect (void);

  This API is used to de-assert the CS (high) to release the SPI bus.

- void LPC17xx_SPI_Release (void);

  This API is used to release the SPI bus.

- void LPC17xx_SPI_SetSpeed (uint8_t speed);

  This API is used to configure the SPI data bit rate.

  During SDC/MMC initialization phase, the speed is normally set to 400 kHz while in data transfer phase; it can be set to a high speed (MMC up to 20 MHz and SDC up to 25 MHz).

- void LPC17xx_SPI_SendByte (uint8_t data);

  This API is used to send one byte of data through SPI bus.

- uint8_t LPC17xx_SPI_RecvByte (void);

  This API is used to receive one byte of data through SPI bus.

2.6 SDC/MMC drivers and APIs

A total of four API functions are provided for accessing SDC/MMC in lpc17xx_sd.c:

- bool LPC17xx_SD_Init (void);

  This API is used to initialize the SDC/MMC.

- bool LPC17xx_SD_ReadCfg (SDCFG *cfg);

  This API is used to read SDC/MMC configuration including register OCR, CID, CSD and some calculated parameters such as sector count, sector size, etc.

- bool LPC17xx_SD_ReadSector (uint32_t sector, uint8_t *buff, uint32_t count);

  This API is used to read specified number of sectors of data from the SDC/MMC. Sector size is fixed to 512 bytes.

- bool LPC17xx_SD_WriteSector (uint32_t sector, const uint8_t *buff, uint32_t count);

  This API is used to write specified number of sectors of data to SDC/MMC.

3. EFSL and FatFs Introduction

3.1 About FAT

The FAT (File Allocation Table) file system (also known as FAT12, FAT16 and FAT32) was developed by Bill Gates and Marc McDonald. It is the primary file system architecture now widely used on most operating systems and memory cards.

FAT was created for managing disks efficiently. The name originates from the usage of a table which centralizes the information about which areas belong to files, are free or possibly unusable, and where each file is stored on the disk. To limit the size of the table, disk space is allocated to files in contiguous groups of hardware sectors called **clusters**. As disk drives have evolved, the maximum number of clusters has dramatically increased, as well as the number of bits used to identify each cluster. The successive major versions of the FAT format are named after the number of table element bits: 12, 16, and 32. The FAT standard has also been expanded in other ways while preserving backward compatibility with existing software.

For more information about FAT and applicable licenses and/or copyrights, please go to

http://www.microsoft.com/whdc/system/platform/firmware/fatgen.mspx

3.2 About EFSL

The Embedded File Systems Library (EFSL) project aims to create a library for file systems, to be used on various embedded systems. Currently EFSL supports the Microsoft FAT file system family. It is EFSL's intent to create pure ANSI C code that compiles on anything that bears the name 'C compiler'.

Adding code for your specific hardware is straightforward; just add code that fetches or writes a 512 byte sector, and the library will do the rest. Existing code can of course be used, own code is only required when you have hardware for which no target exists. For example, it supports secure digital cards in SPI mode.

This project is released under the regular Public License with an exception clause. This clause states that you are allowed to statically link against the library without having to license your own code as GPL as well.

For more information about EFSL and applicable licenses and/or copyrights, please go to

http://efsl.be/

3.3 About FatFs

FatFs is a generic FAT file system module for small embedded systems. The FatFs is written in compliance with ANSI C and is completely separate from the disk I/O layer; it is independent of hardware architecture. It can be incorporated into low cost microcontrollers without any change.

The FatFs has the following features:

- Windows compatible FAT12/16/32 file system.
- Platform independent; easy to port.
- Very small footprint for code and work area.
- Various configuration options:
  - Multiple volumes (physical drives and partitions).
  - Multiple OEM code pages including DBCS.
  - Long File Name (LFN) support in OEM code or Unicode.
  - RTOS support.
  - Multiple sector size support.
  - Read-only, minimized API, I/O buffer, etc.

The FatFs module is free software available for education, research and development. You can use, modify and/or redistribute it for personal, non-profit use or commercial products without any restriction under your responsibility.

For more information about FatFs and applicable licenses and/or copyrights, please go to

http://elm-chan.org/fsw/ff/00index_e.html

4. EFSL port on LPC1700

4.1 EFSL structure

The EFSL internal structure is shown in Fig 3:

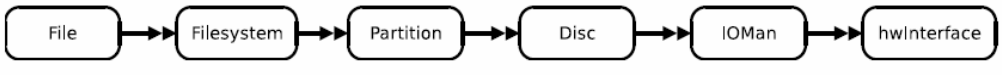


Fig 3. EFSL structure

EFSL has created a linear object model that is quite simple. The ***Filesystem*** object deals with handling the file system specific tasks. The ***Partition*** object is responsible for translating partition relative addressing into disc-based LBA addressing. The ***Disc*** object holds the partition table, and has a direct link to a cache manager, IOMan. In ***IOMan***, all requests for disc sectors come together. IOMan will perform checks to see if sectors have to be read from disc (or from memory), or written back to disc. In the latter case (reading or writing to disc), a request is made to the ***hardware*** layer.

The hardware interface has three responsibilities:

1. Initialize the hardware
2. Read sectors from disc
3. Write sectors to disc

All requests are sector-based. A sector is a 512 byte piece from the disc, which is aligned to a 512 byte boundary.

EFSL port on LPC1700 is rather straightforward; just add code that will fetch or write a 512 byte sector, and the library will do the rest.

The rest of this section will describe step by step how to port EFSL (revision 0.2.8) to LPC1700.

4.2 Setup basic framework

4.2.1 Define a name for your endpoint

You will need this name to create the required defines in the source code. In this project, the name is **HW_ENDPOINT_LPC17xx_SD,** which is defined in config.h:

```
 * Here you will define for what hardware-endpoint EFSL should be compiled.
 * Look in interfaces.h to see what systems are supported, and add your own
 * there if you need to write your own driver. Then, define the name you
 * selected for your hardware there here. Make sure that you only select one
 * device!
*/
    /*#define HW_ENDPOINT_LINUX*/
    /*#define HW_ENDPOINT_ATMEGA128_SD*/
    //#define HW_ENDPOINT_LPC2000_SD
    /* defines the interface for LPC213x (0=SPI0 1=SPI1) */
    // #define HW_ENDPOINT_LPC2000_SPINUM  (0)
    //#define HW_ENDPOINT_LPC2000_SPINUM  (1)
    /*#define HW_ENDPOINT_DSP_TI6713_SD*/
    /* define the interface for LPC17xx SSP0 */
    #define HW_ENDPOINT_LPC17xx_SD
```

Fig 4. Name definition for LPC17xx

4.2.2 Define the sizes of integer types

Open inc/types.h and create a new entry. You may use copy-paste if one of the existing sets is identical to yours.

```
typedef char eint8;
typedef signed char esint8;
typedef unsigned char euint8;
typedef short eint16;
typedef signed short esint16;
typedef unsigned short euint16;
typedef int eint32;
typedef signed int esint32;
typedef unsigned int euint32;
```

Fig 5. Integer types definitions for EFSL

4.2.3 Add your endpoint to interface.h

Add the new entry in inc/interface.h.

```
#if defined(HW_ENDPOINT_LINUX) || defined(HW_ENDPOINT_LINUX64)
    #include "interfaces/linuxfile.h"
#elif defined(HW_ENDPOINT_ATMEGA128_SD)
    #include "interfaces/atmega128.h"
#elif defined(HW_ENDPOINT_DSP_TI6713_SD)
    #include "interfaces/dsp67xx.h"
#elif defined(HW_ENDPOINT_LPC2000_SD)
    #include "interfaces/lpc2000_spi.h"
#elif defined(HW_ENDPOINT_LPC17xx_SD)
    #include "interfaces/if_lpc17xx.h"
#else
    #error "NO INTERFACE DEFINED - see interface.h"
#endif
```

Fig 6. Add endpoint in interface.h

4.2.4 Configure EFSL

The configuration file (\efsl\conf\config.h) defines the behavior of the library. In the configuration files there are many settings, most of which default to safe or standard compliant settings.

The configurations used in this project are listed in Table 3.

Table 3. Configurations of EFSL in this project

Item	Configuration	Description
Hardware target	#define HW_ENDPOINT_LPC17xx_SD	Access SDC/MMC via LPC17xx SSP0
Memory	/* #define BYTE_ALIGNMENT */[1]	Specify that the MCU can not access memory byte oriented
Cache	#define IOMAN_NUMBUFFER 6 #define IOMAN_NUMITERATIONS 3 #define IOMAN_DO_MEMALLOC	6x512 byte (3 KB) RAM used for cache
Cluster pre-allocation	#define CLUSTER_PREALLOC_FILE 2 #define CLUSTER_PREALLOC_DIRECTORY 0	The number of clusters pre-allocated when writing files.

Item	Configuration	Description
Endianess	#define LITTLE_ENDIAN	All FAT structures are stored in Intel little endian order
Date and Time support	/*#define DATE_TIME_SUPPORT*/	Disable date and time support
Error reporting support	#define FULL_ERROR_SUPPORT	Enable error recording for all object
List options	#define LIST_MAXLENFILENAME 12	Configure what kind of data you will get from directory listing requests
Debugging	/* #define DEBUG */	Disable debugging behavior

[1] Being commented out means the macro is not defined.

4.2.5 Create source files

Create header files in inc/interfaces and source files in src/interfaces. In this project, we use lpc17xx_spi.h, lpc17xx_sd.h, lpc17xx_spi.c and lpc17xx_sd.c.

Lpc17xx_spi.c(h) includes APIs to communicate via SSP0 on LPC1700.

Lpc17xx_sd.c(h) includes APIs to access SDC/MMC via SSP0 on LPC1700.

4.3 Implement low level functions

4.3.1 hwInterface

This structure represents the underlying hardware. There are some fields that are required to be present (because EFSL uses them), but you may put in as much or as little as your driver requires to access the hardware.

As always, in embedded design it is recommended to keep this structure as small as possible.

```
/*************************************************************\
                hwInterface
                 ----------
* long      sectorCount     Number of sectors on the file.
\*************************************************************/
struct  hwInterface{
    euint32     sectorCount;
};
typedef struct hwInterface hwInterface;
```

Fig 7. Structure hwInterface

4.3.2 lf_initInterface

This function will be called one time, when the hardware object is initialized by efs_init(). This code should bring the hardware in a ready to use state.

It is recommended, but not required, to fill in the sectorCount filed in structure hwInterface.

```
esint8 if_initInterface(hwInterface* file, eint8* opts)
{
    SDCFG SDCfg;

    if (LPC17xx_SD_Init() == false)
        return (-1);
    if  (LPC17xx_SD_ReadCfg(&SDCfg) == false)
        return (-2);

    file->sectorCount = SDCfg.sectorcnt;

    return 0;
}
```

Fig 8. Implementation of if_initInterface

4.3.3 If_readBuf

This function is used to read a sector from the disc and store it in a user supplied buffer.

Please be very careful to respect the boundaries of the buffers, since it will usually be IOMan calling this function. If you have a buffer overflow, you might corrupt the cache of the next buffer, which may produce extremely rare and impossible to retrace behavior.

```
/*
    read a sector from the disc and store it in a user supplied buffer.

    note that there is no support for sectors that are not 512 bytes large
*/

esint8 if_readBuf(hwInterface* file,euint32 address,euint8* buf)
{
    if (LPC17xx_SD_ReadSector (address, buf, 1) == true)
        return 0;
    else
        return (-1);
}
```

Fig 9. Implementation of if_readBuf

This is an LBA address, relative to the beginning of the disc. When accessing an old hard disc, or a device which uses some other form of addressing, you must recalculate the address to your own addressing scheme. Please note that there is no support for sectors that are not 512 bytes large.

4.3.4 If_writeBuf

The function works exactly the same as its reading variant.

```
/*
    write a sector.

    note that there is no support for sectors that are not 512 bytes large.
*/

esint8 if_writeBuf(hwInterface* file,euint32 address,euint8* buf)
{
    if ( LPC17xx_SD_WriteSector(address, buf, 1) == true)
        return 0;
    else
        return (-1);
}
```

Fig 10. Implementation of if_writeBuf

4.4 Demo

Create a Keil uVision4 project and add all related source files.

Main.c is the test file. It will list all files in the root directory, open a file, and then append one line at the end of that file.

This demo is tested on the KEIL MCB1700 evaluation board. For more information about MCB1700, please refer to: http://www.keil.com/mcb1700/.

Tera term (or other tools) is used for serial communication between PC terminal and MCB1700 and configured at 115200 baud, 8-bits, no parity, 1 stop bit, XON/XOFF.

A 2 GB Kingston Micro SD card is used for test.

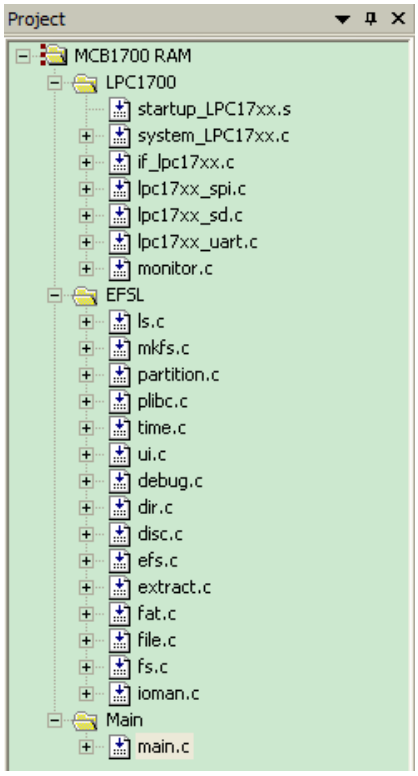
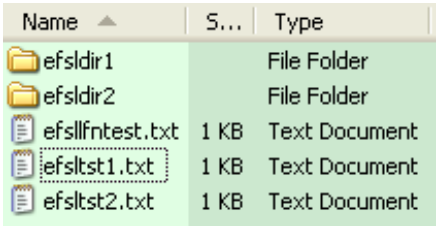


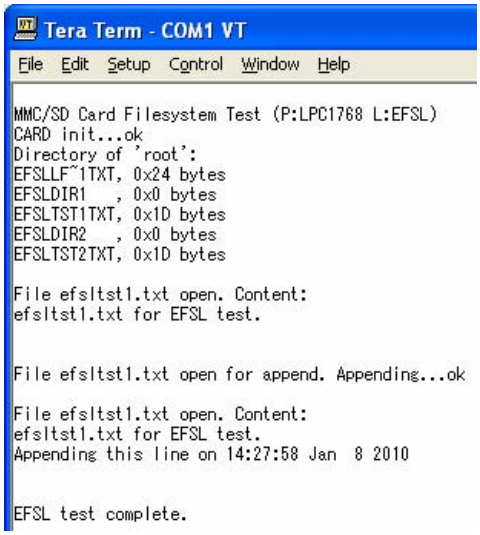
Fig 11. Source files of the project

The file structure in root directory of SDC/MMC and COM output are shown below.

Remark: Since EFSL does not support long file name (LFN), file “efsllfntest.txt” is displayed as “efsllf~1.txt”.



a. Files on root directory



b. COM output

Fig 12. Demo result

5. FatFs port on LPC1700

5.1 FatFs structure

The FatFs structure is shown in Fig 13.

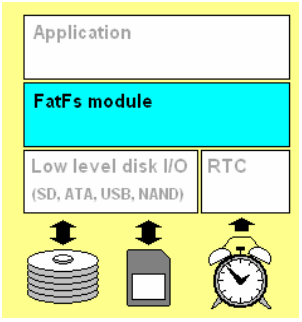


Fig 13. FatFs structure

FatFs module is a middleware which provides many functions to access the FAT volumes, such as f_open, f_close, f_read, f_write, etc (refer to ff.c). There is no platform dependence in this module, as long as the compiler is in compliance with ANSI C.

A low level disk I/O module is used to read/writ the physical disk.

An RTC module is used to get the current time.

The Low level disk I/O and RTC module are completely separate from the FatFs module. They must be provided by the user, which is the main task of porting FatFs module to other platform.

The rest of this section will describe step by step how to port FatFs (revision 0.07e) to LPC1700.

5.2 Define the size of integer types

The FatFs module assumes that the size of ***char/short/long*** are 8/16/32-bit and ***int*** is 16 or 32 bit. These correspondences are defined in integer.h. This will not be a problem on most compilers. Any conflict with existing definitions must be resolved carefully.

```
/* These types must be 16-bit, 32-bit or larger integer */
typedef int             INT;
typedef unsigned int    UINT;

/* These types must be 8-bit integer */
typedef signed char     CHAR;
typedef unsigned char   UCHAR;
typedef unsigned char   BYTE;

/* These types must be 16-bit integer */
typedef short           SHORT;
typedef unsigned short  USHORT;
typedef unsigned short  WORD;
typedef unsigned short  WCHAR;

/* These types must be 32-bit integer */
typedef long            LONG;
typedef unsigned long   ULONG;
typedef unsigned long   DWORD;
```

Fig 14. Integer types definitions for FatFs module

5.3 Configure the FatFs module

All of the configurations and detailed descriptions can be found in ffconf.h (for FatFs revision 0.07e).

The configurations used in this project are listed in Table 4.

Table 4. Configurations of FatFs module in this project

Item	Configuration	Description
Function and Buffer Configurations	#define _FS_TINY 0	Use the sector buffer in the individual file data transfer.
	#define _FS_READONLY 0	Enable both read and write functions.
	#define _FS_MINIMIZE 0	Enable full function.
	#define _USE_STRFUNC 0	Disable string functions.
	#define _USE_MKFS 1	Enable f_mkfs function
	#define _USE_FORWARD 0	Disable f_forward function
Locale and Namespace Configurations	#define _CODE_PAGE 850	OEM code page “Multilingual Latin 1” will be used on the target system.
	#define _USE_LFN 1	Enable LFN
	#define _MAX_LFN 255	Maximum LFN length to handle
	#define _LFN_UNICODE 0	Disable Unicode.
	#define _FS_RPATH 1	Enable the relative path feature and f_chdir and f_chdrive function are available.
Physical Drive Configurations	#define _DRIVES 1	Only 1 physical driver is allowed.
	#define _MAX_SS 512	Maximum sector size to be handled

Item	Configuration	Description
	#define _MULTI_PARTITION 0	Each volume is bound to the same physical drive number and can mount only first primary partition.
System Configurations	#define _WORD_ACCESS 0	Enable the Byte-by-byte access
	#define _FS_REENTRANT 0	Disable reentrancy.

5.3.1 _USE_LFN

The FatFs module supports Long File Name (LFN) in revision 0.07e. The two different file names, SFN and LFN, of a file are transparent in the file functions except for f_readdir function. To enable LFN feature, set _USE_LFN to 1 or 2, and add a Unicode code conversion function ff_convert and ff_wtoupper to the project. This function is available in option\cc*.c.

Note that the LFN feature on the FAT file system is a patent of Microsoft Corporation. When enabled on commercial products, a license from Microsoft may be required depending on the final destination.

5.3.2 _CODE_PAGE

The _CODE_PAGE specifies the OEM code page to be used on the target system. Incorrect setting of the code page can cause a file open failure.

When the LFN feature is enabled, the module size will be increased depending on the selected code page. Table 5 shows the difference in module size when LFN is enabled with some code pages. The Chinese and Korean language have tens of thousands of characters which require a huge OEM-Unicode bidirectional conversion table; therefore, the module size will be drastically increased as shown in Table 5. As a result, the FatFs with LFN will not be able to be implemented in some microcontrollers with limited ROM size.

Table 5. ROM size increase with different code pages on Cortex-M3

Code page	ROM size increase (byte)
SBSC	2796
CP932 (Japanese Shift-JIS)	61656
CP936 (Simplified Chinese GBK)	176856
CP949 (Korean)	138912
CP950 (Traditional Chinese Big5)	110544

[1] Compiler: armcc V4.0.0 Optimization: O3

5.4 Implement low level functions

Since the FatFs module is completely separate from the disk I/O and RTC module, it requires the following functions to read/write the physical disk and to get the current time. Because the low level disk I/O and RTC module are not a part of the FatFs module, they must be provided by the user.

5.4.1 disk_initialize

The disk_initialize function initializes a physical drive.

This function is called from the volume mount process in the FatFs module to manage the media change. The application program must not call this function while the FatFs module is active; the FAT structure on the volume may collapse. To re-initialize the file system, use f_mount function.

```
/*-----------------------------------------------------------------------*/
/* Initialize Disk Drive                                                 */
/*-----------------------------------------------------------------------*/
DSTATUS disk_initialize (
    BYTE drv        /* Physical drive number (0) */
)
{
    if (drv) return STA_NOINIT;         /* Supports only single drive */
//  if (Stat & STA_NODISK) return Stat; /* No card in the socket */

    if (LPC17xx_SD_Init() && LPC17xx_SD_ReadCfg(&SDCfg))
        Stat &= ~STA_NOINIT;

    return Stat;
}
```

Fig 15. Implementation of disk_initialize

5.4.2 disk_status

The disk_status function returns the current disk status which is a combination of the following flags.

- STA_NOINIT: Indicates that the disk drive has not been initialized.
- STA_NODISK: Indicates that no medium is in the drive.
- STA_PROTECTED: Indicates that the medium is write protected.

Since the MCB1700 board does not provide card detection and write protection, we will neglect these two flags: STA_NODISK and STA_PROTECTED.

```
/*-----------------------------------------------------------------------
/* Get Disk Status
/*-----------------------------------------------------------------------
DSTATUS disk_status (
    BYTE drv        /* Physical drive number (0) */
)
{
    if (drv) return STA_NOINIT;     /* Supports only single drive */

    return Stat;
}
```

Fig 16. Implementation of disk_status

5.4.3 disk_read

The disk_read function reads one or more sectors from the disk drive.

```
/*-----------------------------------------------------------------------*/
/* Read Sector(s)                                                        */
/*-----------------------------------------------------------------------*/
DRESULT disk_read (
    BYTE drv,           /* Physical drive number (0) */
    BYTE *buff,         /* Pointer to the data buffer to store read data */
    DWORD sector,       /* Start sector number (LBA) */
    BYTE count          /* Sector count (1..255) */
)
{
    if (drv || !count) return RES_PARERR;
    if (Stat & STA_NOINIT) return RES_NOTRDY;

    if (LPC17xx_SD_ReadSector (sector, buff, count) == true)
        return RES_OK;
    else
        return RES_ERROR;
}
```

Fig 17. Implementation of disk_read

5.4.4 disk_write

The disk_write function writes one or more sectors to the disk.

This function is not required in read only configuration.

```
/*-----------------------------------------------------------------------
/* Write Sector(s)
/*-----------------------------------------------------------------------
#if _READONLY == 0
DRESULT disk_write (
    BYTE drv,           /* Physical drive number (0) */
    const BYTE *buff,   /* Pointer to the data to be written */
    DWORD sector,       /* Start sector number (LBA) */
    BYTE count          /* Sector count (1..255) */
)
{
    if (drv || !count) return RES_PARERR;
    if (Stat & STA_NOINIT) return RES_NOTRDY;
//  if (Stat & STA_PROTECT) return RES_WRPRT;

    if ( LPC17xx_SD_WriteSector(sector, buff, count) == true)
        return RES_OK;
    else
        return  RES_ERROR;

}
#endif /*  READONLY == 0 */
```

Fig 18. Implementation of disk_write

5.4.5 disk_ioctl

The disk_ioctl function controls device specified features and miscellaneous functions other than disk read/write.

Table 6. Supported commands in disk_ioctl functions

Command	Description
Device independent	
CTRL_SYNC	Ensures that the disk drive has finished pending write process. When the disk I/O module has a write back cache, flush the dirty sector immediately. This command is not required in read-only configuration
GET_SECTOR_SIZE	Returns sector size of the drive into the WORD variable pointed by Buffer. This command is not required in single sector size configuration, _MAX_SS is 512.
GET_SECTOR_COUNT	Returns total sectors on the drive into the DWORD variable pointed by Buffer. This command is used in only f_mkfs function.
GET_BLOCK_SIZE	Returns erase block size of the memory array in unit of sector into the DWORD variable pointed by Buffer. This command is used in only f_mkfs function.
Device dependent	
MMC_GET_TYPE	Get card type flags (1 byte)
MMC_GET_CSD	Receive CSD as a data block (16 bytes)
MMC_GET_CID	Receive CID as a data block (16 bytes)
MMC_GET_OCR	Receive OCR as an R3 response (4 bytes)
MMC_GET_SDSTAT	Receive SD status as a data block (64 bytes)

Please refer to the software example for the detailed implementation of these functions.

5.4.6 get_fattime

The get_fattime function gets current time which is not required in read only configuration.

```
/*---------------------------------------------------------*/
/* User Provided RTC Function for FatFs module             */
/*---------------------------------------------------------*/
/* This is a real time clock service to be called from     */
/* FatFs module. Any valid time must be returned even if   */
/* the system does not support an RTC.                     */
/* This function is not required in read-only cfg.         */
DWORD get_fattime ()
{
    RTCTime rtc;

    /* Get local time */
    rtc_gettime(&rtc);

    /* Pack date and time into a DWORD variable */
    return    ((DWORD)(rtc.RTC_Year - 1980) << 25)
            | ((DWORD)rtc.RTC_Mon << 21)
            | ((DWORD)rtc.RTC_Mday << 16)
            | ((DWORD)rtc.RTC_Hour << 11)
            | ((DWORD)rtc.RTC_Min << 5)
            | ((DWORD)rtc.RTC_Sec >> 1);
}
```

Fig 19. Implementation of get_fattime

5.5 Demo

This demo is also tested on Keil's MCB1700 evaluation board with the same 2 GB Kingston Micro SD card and COM configuration.

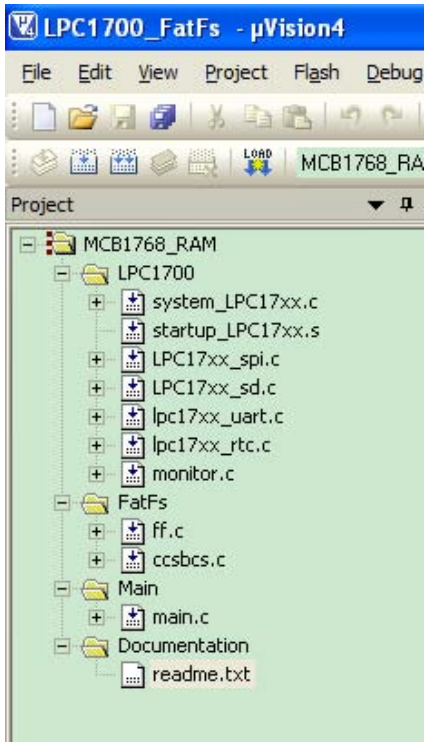
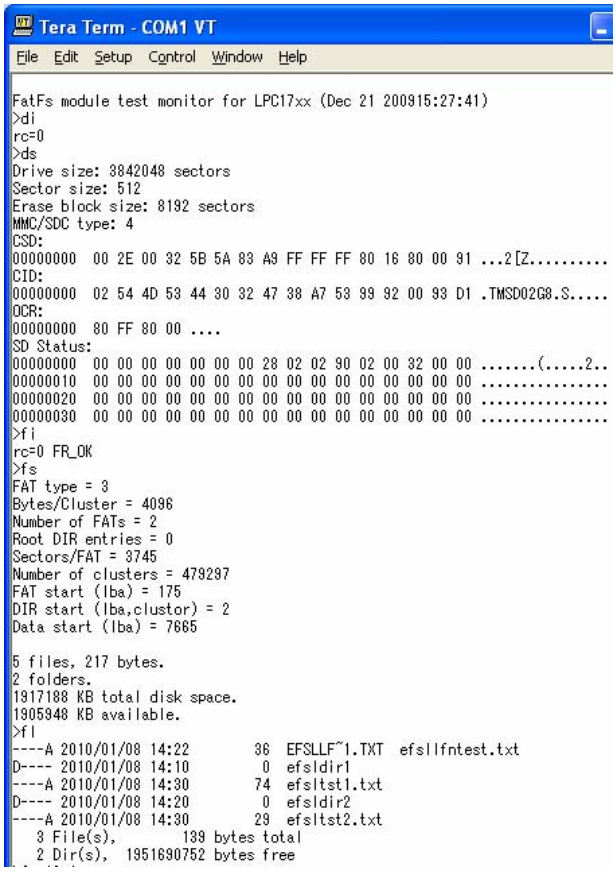


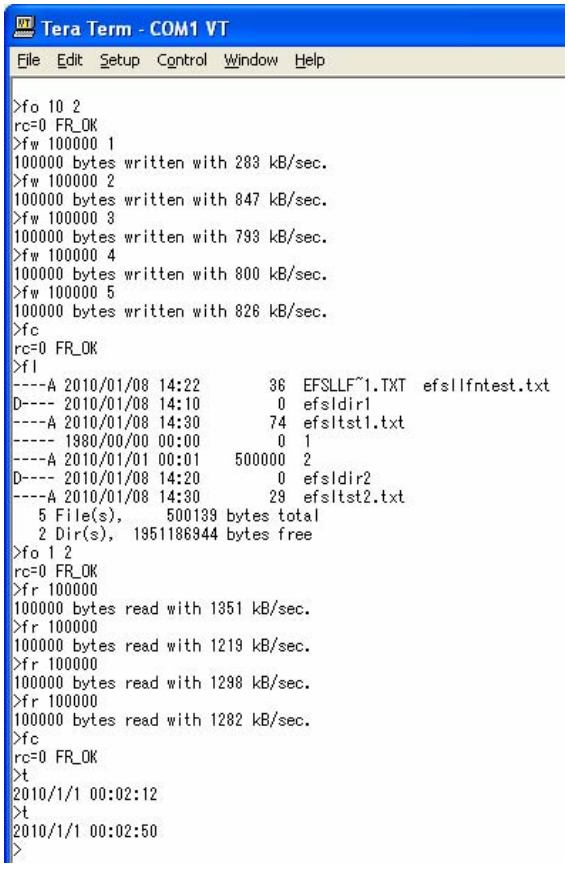
Fig 20. Source files of the project

Table 7. Supported commands for FatFs

Command	Description
Disk functions	
Di	Initialize the disk
Ds	Show disk status
dd [<lba>]	Dump a specific sector
File functions	
fi	Force initialize the logical drive
fs	Show logical drive status
fl [<path>]	Directory listing
fo <mode> <file>	Open a file
fc	Close a file
fe	Seek file pointers
fd <len>	Read and dump file from current fp
fr <len>	Read file
fw <len> <val>	Write file
fn <old_name> <new_name>	Change file/dir name
fu <name>	Unlink a file or dir
fv	Truncate file
fk <name>	Create a directory
fa <atrr> <mask> <name>	Change file/dir attribute
ft <year> <month> <day> <hour> <min> <sec> <name>	Change timestamp
fx <src_name> <dst_name>	Copy file
fg <path>	Change current directory
fj <drive#>	Change current drive
fm <partition rule> <cluster size>	Create file system
fz [<rw size>]	Change R/W length for fr/fw/fx command
Time functions	
t [<year> <mon> <mday> <hour> <min> <sec>]	Get or set the current date and time



a. Card and file system information test



b. Card R/W and RTC test

Fig 21. Demo output

6. References

[1] NXP LPC17xx User Manual UM10360 (Rev. 00.07), NXP Semiconductors, July 31, 2009

[2] SanDisk SD Card Product Manual (Version 2.2), SanDisk Corporation. Nov, 2004

[3] The MultiMediaCard System Specification, Version 3.1, MMC Association Technical Committee, June 2001.

7. Legal information

7.1 Definitions

Draft — The document is a draft version only. The content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included herein and shall have no liability for the consequences of use of such information.

7.2 Disclaimers

Limited warranty and liability — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms and conditions of commercial sale of NXP Semiconductors.

Right to make changes — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Suitability for use — NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of an NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors accepts no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer's own risk.

Applications — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

Export control — This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from national authorities.

In no event shall NXP Semiconductors, its affiliates or their suppliers be liable to customer for any special, indirect, consequential, punitive or incidental damages (including without limitation damages for loss of business, business interruption, loss of use, loss of data or information, and the like) arising out the use of or inability to use the product, whether or not based on tort (including negligence), strict liability, breach of contract, breach of warranty or any other theory, even if advised of the possibility of such damages.

Notwithstanding any damages that customer might incur for any reason whatsoever (including without limitation, all damages referenced above and all direct or general damages), the entire liability of NXP Semiconductors, its affiliates and their suppliers and customer's exclusive remedy for all of the foregoing shall be limited to actual damages incurred by customer based on reasonable reliance up to the greater of the amount actually paid by customer for the product or five dollars (US$5.00). The foregoing limitations, exclusions and disclaimers shall apply to the maximum extent permitted by applicable law, even if any remedy fails of its essential purpose.

7.3 Trademarks

Notice: All referenced brands, product names, service names and trademarks are property of their respective owners.

8. Contents

For more information, please visit: http://www.nxp.com
For sales office addresses, please send an please send an email to:
salesaddresses@nxp.com

Date of release: 6 July 2010
Document identifier: AN10916