FatFs Module Application Note

How to Port

Basic Considerations

The FatFs module assumes following conditions on portability.

- ANSI C
  The FatFs module is a middleware written in ANSI C (C89). There is no platform dependence, so long as the compiler is in compliance with C89 or later. Only exFAT feature requires C99.
- Size of integer types
  - Size of `char` must be 8-bit.
  - Size of `int`, as well as integer promotion, must be 16-bit or 32-bit.
  - When the C standard is in C89, size of `short` and `long` must be 16-bit and 32-bit respectively.
  - When it is in C99 or later, `stdint.h` is used to obtain the integer sizes.

Integer Types in FatFs API

Integer types used in FatFs are defined in `ff.h` as described below. It is based on Win32 API (`windef.h`). This will not be a problem on most platform. When a conflict with existing definitions occured, you must resolve it with care.

`BYTE`
: 8-bit unsigned integer in range of 0 to 2^8 - 1.

`WORD`
: 16-bit unsigned integer in range of 0 to 2^{16} - 1.

`DWORD`
: 32-bit unsigned integer in range of 0 to 2^{32} - 1.

`QWORD`
: 64-bit unsigned integer in range of 0 to 2^{64} - 1.

`UINT`
: Alias of `unsigned int` used to specify any number.

`WCHAR`
: Alias of `WORD` used to specify a UTF-16 code unit.

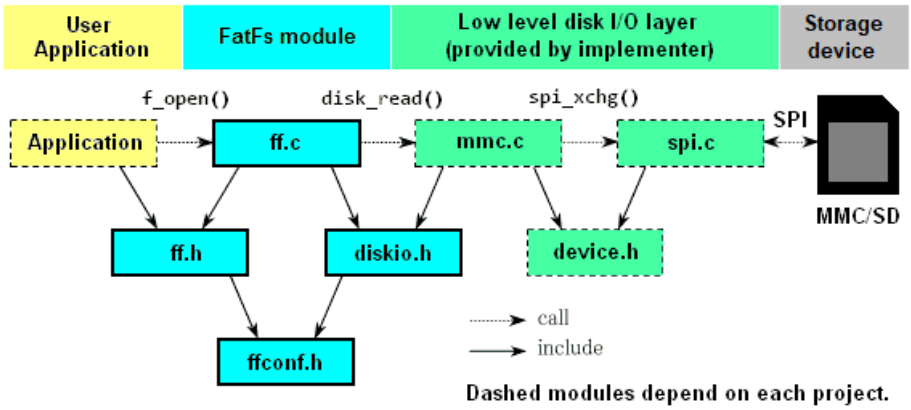
`TCHAR`
: Alias of `char`, `WCHAR` or `DWORD` used to specify a character encoding unit.

`FSIZE_t`
: Alias of `DWORD` or `QWORD` used to address file offset and to specify file size.

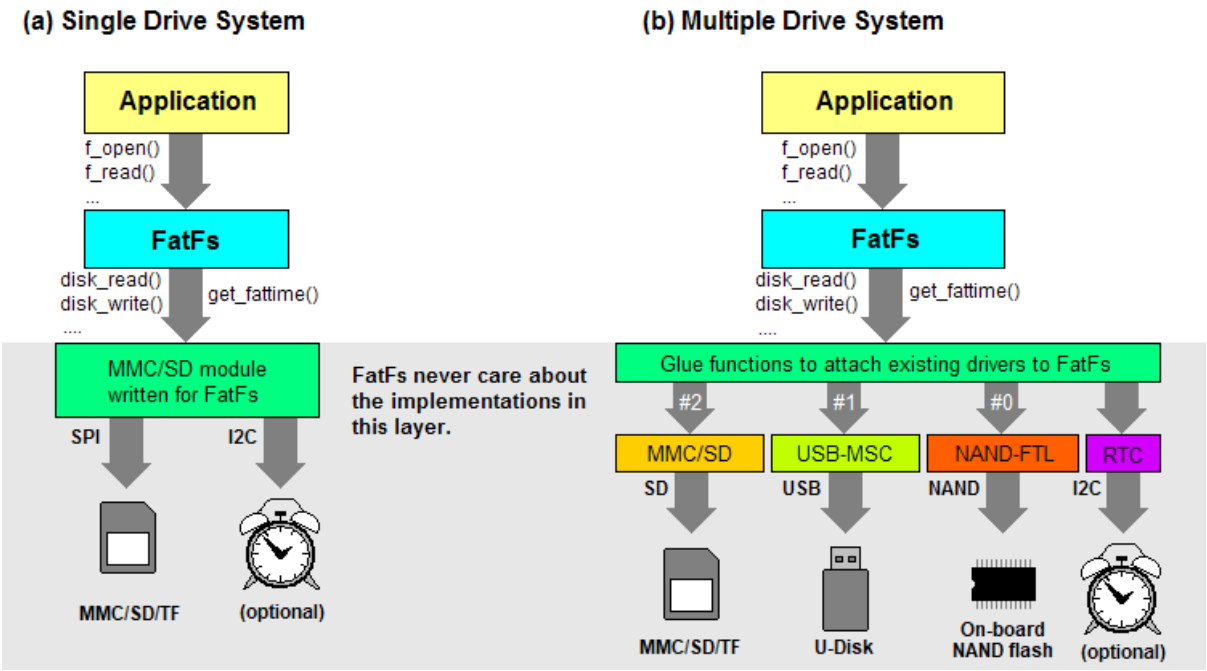
`LBA_t`
: Alias of `DWORD` or `QWORD` used to address sectors in LBA and to specify number of sectors.

System Organizations

The dependency diagram shown below is a typical, but not specific, configuration of the embedded system with FatFs module.



(a) If a working disk module for FatFs is provided, nothing else will be needed. (b) To attach existing disk drivers with different interface, some glue functions are needed to translate the interfaces between FatFs and the driver.



Required Functions

You need to provide only MAI functions required by FatFs module and nothing else. If any working device control module for the target system is available, you need to write only glue functions to attach it to the FatFs module. If not, you need to port another device control module or write it from scratch. Most of MAI functions are not that always required. For instance, any write function is not required in read-only configuration. Following table shows which function is required depends on the configuration options.

Function	Required when	Note
disk_status disk_initialize disk_read	Always	
disk_write get_fattime disk_ioctl (CTRL_SYNC)	FF_FS_READONLY == 0	Disk I/O functions. Samples available in ffsample.zip. There are many implementations on the web.
disk_ioctl (GET_SECTOR_COUNT) disk_ioctl (GET_BLOCK_SIZE)	FF_USE_MKFS == 1	
disk_ioctl (GET_SECTOR_SIZE)	FF_MAX_SS != FF_MIN_SS	
disk_ioctl (CTRL_TRIM)	FF_USE_TRIM == 1	
ff_uni2oem ff_oem2uni ff_wtoupper	FF_USE_LFN != 0	Unicode support functions. Add optional module ffunicode.c to the project.
ff_cre_syncobj ff_del_syncobj ff_req_grant ff_rel_grant	FF_FS_REENTRANT == 1	O/S dependent functions. Sample code is available in ffsystem.c.
ff_mem_alloc	FF_USE_LFN == 3	

ff_mem_free

FatFs cares about neither what kind of storage device is used nor how it is implemented. Only a requirement is that it is a block device read/written in fixed-size blocks that accessible via the disk I/O functions defined above.

Limits

- Filesystem type: FAT, FAT32(rev0.0) and exFAT(rev1.0).
- Number of open files: Unlimited. (depends on available memory)
- Number of volumes: Up to 10.
- Sector size: 512, 1024, 2048 and 4096 bytes.
- Minimum volume size: 128 sectors.
- Maximum volume size: 2^{32} - 1 sectors in 32-bit LBA, virtually unlimited in 64-bit LBA with exFAT.
- Maximum file size: 2^{32} - 1 bytes on FAT volume, virtually unlimited on exFAT volume.
- Cluster size: Upto 128 sectors on FAT volume and up to 16 MB on exFAT volume.

Memory Usage

The memory usage varies depends on the configuration options.

	ARM7 32bit	**ARM7 Thumb**	**CM3 Thumb-2**	**AVR**	**H8/300H**	**PIC24**	**RL78**	**V850ES**	**SH-2A**	**RX600**	**IA-32**
Compiler	GCC	GCC	GCC	GCC	CH38	C30	CC78K0R	CA850	SHC	RXC	MSC
text (Full, R/W)	10.6k	6.8k	6.4k	12.4k	10.1k	11.4k	13.1k	8.9k	9.1k	6.5k	9.0k
text (Min, R/W)	7.2k	4.8k	4.6k	8.5k	7.1k	8.0k	9.6k	6.3k	6.3k	4.7k	6.4k
text (Full, R/O)	5.0k	3.2k	2.9k	6.0k	4.8k	5.5k	6.6k	4.4k	4.1k	3.2k	4.3k
text (Min, R/O)	3.8k	2.5k	2.3k	4.4k	3.8k	4.2k	5.2k	3.4k	3.3k	2.5k	3.5k
bss	V*4 + 2	V*4 + 2	V*4 + 2	V*2 + 2	V*4 + 2	V*2 + 2	V*2 + 2	V*4 + 2	V*4 + 2	V*4 + 2	V*4 + 2
Work area (FF_FS_TINY == 0)	V*564 + F*552	V*564 + F*552	V*564 + F*552	V*560 + F*546	V*560 + F*546	V*560 + F*546	V*560 + F*546	V*564 + F*552	V*564 + F*552	V*564 + F*552	V*564 + F*552
Work area (FF_FS_TINY == 1)	V*564 + F*40	V*564 + F*40	V*564 + F*40	V*560 + F*34	V*560 + F*34	V*560 + F*34	V*560 + F*34	V*564 + F*40	V*564 + F*40	V*564 + F*40	V*564 + F*40

These are the memory usage of FatFs module without lower layer on some target systems in following condition. *V* denotes number of mounted volumes and *F* denotes number of open files. Every samples here are optimezed in code size.

```
FatFs R0.14a options:
FF_FS_READONLY   0 (Read/Write) or 1 (Read only)
FF_FS_MINIMIZE   0 (Full, with all basic functions) or 3 (Min, with fully minimized)
FF_FS_TINY       0 (Default) or 1 (Tiny file object)
And any other options are left unchanged from original setting.
```

Reducing Module Size

Follwing table shows which API function is removed by configuration options for the module size reduction. To use any API function, the row of the function must be clear.

Function	FF_FS_MINIMIZE				FF_FS_READONLY		FF_USE_STRFUNC		FF_FS_RPATH			FF_USE_FIND		FF_USE_CHMOD		FF_USE_EXPAND		FF_USE_LABEL		FF_USE_MKFS		FF_USE_FORWARD		FF_MULTI_PARTITION	
	0	1	2	3	0	1	0	1	0	1	2	0	1	0	1	0	1	0	1	0	1	0	1	0	1
f_mount																									
f_open																									
f_close																									
f_read																									
f_write						x																			
f_sync						x																			
f_lseek				x																					
f_opendir			x	x																					
f_closedir			x	x																					
f_readdir			x	x																					
f_findfirst			x	x								x													
f_findnext			x	x								x													
f_stat	x		x	x																					
f_getfree	x		x	x		x																			
f_truncate	x		x	x		x																			
f_unlink	x		x	x		x																			

f_mkdir	x	x	x	x									
f_rename	x	x	x	x									
f_chdir						x							
f_chdrive						x							
f_getcwd						x	x						
f_chmod				x				x					
f_utime				x				x					
f_getlabel										x			
f_setlabel				x						x			
f_expand				x					x				
f_forward												x	
f_mkfs				x							x		
f_fdisk				x							x		x
f_putc				x	x								
f_puts				x	x								
f_printf				x	x								
f_gets					x								

Long File Name

FatFs module supports the long file name (LFN) extension of the FAT filesystem. The two different file names, short file name (SFN) and LFN, of a file is transparent on the API. The support for LFN feature is disabled by default. To enable the LFN, set `FF_USE_LFN` to 1, 2 or 3, and add `ffunicode.c` to the project. The LFN feature requiers a certain working buffer. The buffer size can be configured by `FF_MAX_LFN` according to the available memory. The length of an LFN can be up to 255 characters, so that the `FF_MAX_LFN` should be set to 255 for all existing file names. If the size of working buffer is insufficient for the input file name, the file function fails with `FR_INVALID_NAME`. When use any re-entry to the API with LFN feature, `FF_USE_LFN` must be set to 2 or 3. In this case, the file function allocates the working buffer on the stack or heap. The LFN working buffer occupies `(FF_MAX_LFN + 1) * 2` bytes and additional `(FF_MAX_LFN + 44) / 15 * 32` bytes when exFAT is enabled.

Impact upon Module Size

With LFN at CM3 + gcc

FF_CODE_PAGE	Code size
437-869 (SBCS)	+3.3k
932 (Japanese)	+62k
936 (Simplified Chinese)	+177k
949 (Korean)	+140k
950 (Traditional Chinese)	+111k
0 (All code pages)	+486k

When the LFN is enabled, the module size will be increased depends on the configured code page. Right table shows the increment of code size in some code pages. Especially, in the CJK region, tens of thousands of characters are being used. Unfortunately, it requires a huge OEM-Unicode bidirectional conversion table and the module size will be drastically increased as shown in the table.

As the result, the FatFs with LFN enabled with DBCS code pages will not able to be ported on the most 8-bit MCU systems. If the target system is in legacy-free, in only Unicode and any ANSI/OEM code is not used at all, the code page setting gets meaningless. You will able to reduce the code size by configureing FatFs for Unicode with any SBCS code page.

There ware some restrictions on using LFN for open source project, because the LFN extension on the FAT filesystem was a patent of Microsoft Corporation. However the related patents all have expired and using the LFN feature is free for any projects.

Unicode API

By default, FatFs uses ANSI/OEM code set on the API even in LFN configuration. FatFs can also switch the character encoding on the API to Unicode by configuration option `FF_LFN_UNICODE`. This means that FatFs is compliant with the full featured LFN specification. The data type `TCHAR` specifies path name strings on the API is an alias of either `char`(ANSI/OEM or UTF-8), `WCHAR`(UTF-16) or `DWORD`(UTF-32) depends on that option. For more information, refer to the description in the file name.

Note that setting of code page, `FF_CODE_PAGE`, has actually no meaning when FatFs is configured for the Unicode API. It should be set 437 anyway. However it still affects code conversion of string I/O functions at `FF_STRF_ENCODE == 0` and backward compatibility with legacy systems, so that code page may need to be configured properly if it is considered a problem.

exFAT Filesystem

The exFAT (Microsoft's Extended File Allocation Table) filesystem is a succession of the FAT/FAT32 filesystem which has been widely used in embedded systems, consumer devices and portable storage media. It is adopted by SDA (SD Association) as the filesystem for

SDXC card, 64 GB and larger, and they are being shipped with this format. Therefore the exFAT is one of the standard filesystems for removable media as well as FAT. The exFAT filesystem allows the file size beyond the 4 GB limit what FAT filesystem allows up to and some filesystem overhead, especially cluster allocation delay, are reduced as well. These features allow to record the large data without dividing into some files and improve the write throughput to the file.

Note that the exFAT filesystem is a patent of Microsoft Corporation. The exFAT feature of FatFs is an implementation based on *US. Pat. App. Pub. No. 2009/0164440 A1*. FatFs module can switch the exFAT on or off by a configuration option, FF_FS_EXFAT. When enable the exFAT for the commercial products, a license by Microsoft will be needed depends on the final destination of the products.

Remarks: Enabling exFAT discards C89 compatibility and it wants C99 because of need for 64-bit integer type.

64-bit LBA

LBA (Logical Block Addressing) is an addressing method to specify the location of data block, called sector, on the storage media. It is a simple linear address beginning from 0 as the first block of data. The host system does not need to consider how the data block is located and managed in the storage device. FatFs supports only LBA for the media access. 32-bit LBA is a common size at the interface of the most storage devices. It can address up to 2^{32} blocks, 2 TB in 512 bytes/sector. When a storage device larger than 2 TB is used, larger sector size or 64-bit LBA will be needed to address the entire block of the storage device.

By default, FatFs works in 32-bit LBA for media access interface. FatFs can also switch it to 64-bit LBA by a configuration option FF_LBA64. It also enables GPT (GUID Partition Table) for partiotion management on the storage device. For further information about GPT, refer to f_mkfs and f_fdisk function.

Re-entrancy

The file operations of two tasks to the *different volumes* each other is always re-entrant regardless of the configurations except when LFN is enabled with static working buffer (FF_USE_LFN = 1). It can work concurrently without any mutual exclusion.

The file operations of two tasks to the *same volume* is not re-entrant in default. FatFs can also be configured to make it thread-safe by option FF_FS_REENTRANT. In this case, also the OS dependent synchronization control functions, ff_cre_syncobj/ff_del_syncobj/ff_req_grant/ff_rel_grant, need to be added to the project. There are some examples in the ffsystem.c. When a file function is called while the volume is being accessed by another task, the file function to the volume will be suspended until that task leaves the file function. If the wait time exceeded a period defined by FF_TIMEOUT, the file function will abort with FR_TIMEOUT. The timeout feature might not be supported on the some RTOSs.

There is an exception on the re-entrancy for f_mount/f_mkfs function. These volume management functions are not re-entrant to the same volume. When use these functions, other tasks need to avoid to access the volume.

Function	Case 1	Case 2	Case 3
disk_status	Yes	Yes	Yes(*)
disk_initialize	No	Yes	Yes(*)
disk_read	No	Yes	Yes(*)
disk_write	No	Yes	Yes(*)
disk_ioctl	No	Yes	Yes(*)
get_fattime	No	Yes	Yes

Case 1: Same volume.
Case 2: Different volume on the same drive.
Case 3: Different volume on the different drive.
(*) In only different drive number.

Remarks: This section describes on the re-entrancy of the FatFs module itself. The FF_FS_REENTRANT option enables only exclusive use of each filesystem objects and FatFs does not that prevent to re-enter the storage device control functions. Thus the device control layer needs to be always thread-safe when FatFs API is re-entered for different volumes. Right table shows which control function can be re-entered when FatFs API is re-entered on some conditions.

Duplicated File Open

FatFs module does not support the read/write collision control of duplicated open to a file. The duplicated open is permitted only when each of open method to a file is read mode. The duplicated open with one or more write mode to a file is always prohibited, and also open file must not be renamed or deleted. A violation of these rules can cause data collaption.

The file lock control can be enabled by FF_FS_LOCK option. The value of option defines the number of open objects to manage simultaneously. In this case, if any opening, renaming or removing against the file shareing rule that described above is attempted, the file function will be rejected with FR_LOCKED. If number of open objects, files and sub-directories, is equal to FF_FS_LOCK, an extra f_open/f_opendir function will fail with FR_TOO_MANY_OPEN_FILES.

Performance Effective File Access

For good read/write throughput on the small embedded systems with limited size of memory, application programmer should consider what process is done in the FatFs module. The file data on the volume is transferred in following sequence by f_read function.

Figure 1. Sector unaligned read (short)

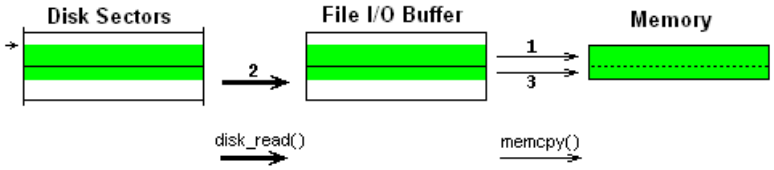


Figure 2. Sector unaligned read (long)

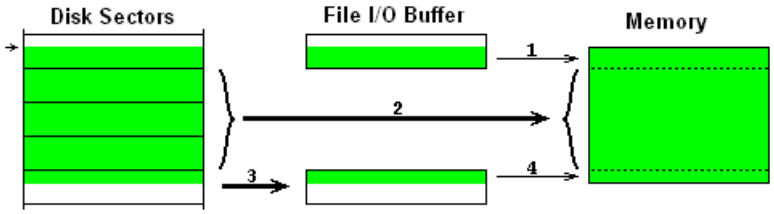
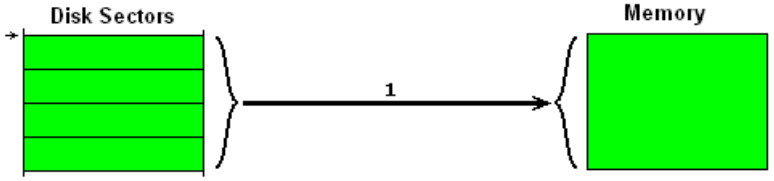


Figure 3. Fully sector aligned read



The file I/O buffer is a sector buffer to read/write a part of data on the sector. The sector buffer is either file private sector buffer on each file object or shared sector buffer in the filesystem object. The buffer configuration option `FF_FS_TINY` determins which sector buffer is used for the file data transfer. When tiny buffer configuration (1) is selected, data memory consumption is reduced `FF_MAX_SS` bytes each file object. In this case, FatFs module uses only a sector buffer in the filesystem object for file data transfer and FAT/directory access. The disadvantage of the tiny buffer configuration is: the FAT data cached in the sector buffer will be lost by file data transfer and it must be reloaded at every cluster boundary. However it will be suitable for most application from view point of the decent performance and low memory comsumption.

Figure 1 shows that a partial sector, sector unaligned part of the file, is transferred via the file I/O buffer. At long data transfer shown in Figure 2, middle of transfer data that covers one or more sector is transferred to the application buffer directly. Figure 3 shows that the case of entier transfer data is aligned to the sector boundary. In this case, file I/O buffer is not used. On the direct transfer, the maximum extent of sectors are read with `disk_read` function at a time but the multiple sector transfer is divided at cluster boundary even if it is contiguous.

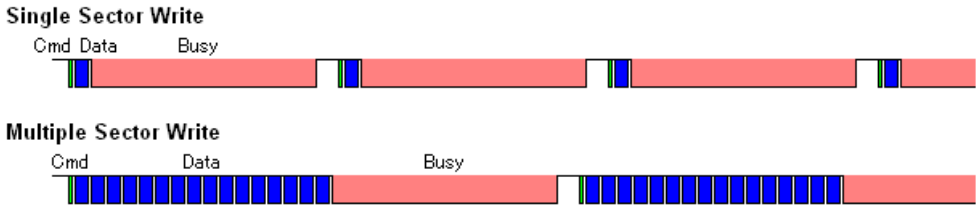
Therefore taking effort to sector aligned read/write accesss eliminates buffered data transfer and the read/write performance will be improved. Besides the effect, cached FAT data will not be flushed by file data transfer at the tiny configuration, so that it can achieve same performance as non-tiny configuration with small memory footprint.

Considerations on Flash Memory Media

To maximize the write performance of flash memory media, such as SDC, CFC and U Disk, it must be controlled in consideration of its characteristitcs.

Using Mutiple-Sector Write

Figure 6. Comparison between Multiple/Single Sector Write



The write throughput of the flash memory media becomes the worst at single sector write transaction. The write throughput increases as the number of sectors per a write transaction as shown in Figure 6. This effect more appers at faster interface speed and the performance ratio often becomes grater than ten. This result is clearly explaining how fast is multiple block write (W:16K, 32 sectors) than single block write (W:100, 1 sector), and also larger card tends to be slow at single block write. Number of write transactions also affects life time of the flash memory media. When compared at same amount of write data, the single sector write in Figure 6 above wears flash memory media 16 times more than multiple sector write in Figure 6 below. Single sector write is pretty pain for the flash memory media.

Therefore the application program should write the data in large block as possible. The ideal write chunk size and alighment is size of sector, and size of cluster is the best. Of course all layers between the application and the storage device must have consideration on multiple sector write, however most of open-source memory card drivers lack it. Do not split a multiple sector write request into single

sector write transactions or the write throughput gets poor. Note that FatFs module and its sample disk drivers supprt multiple sector read/write operation.

Forcing Memory Erase

When remove a file with `f_unlink` function, the data clusters occupied by the file are marked 'free' on the FAT. But the data sectors containing the file data are not that applied any process, so that the file data left occupies a part of the flash memory array as 'live block'. If the file data can be erased on removing the file, those data blocks will be turned into the free block pool. This may skip internal block erase operation to the data block on next write operation. As the result the write performance might be improved. FatFs can manage this function by setting `FF_USE_TRIM` to 1. Note that this is an expectation of internal process of the storage device and not that always effective. Most applications will not need this function. Also `f_unlink` function can take a time when remove a large file.

Critical Section

If a write operation to the FAT volume is interrupted due to an accidental failure, such as sudden blackout, wrong media removal and unrecoverable disk error, the FAT structure on the volume can be broken. Following images shows the critical section of the FatFs module.

Figure 4. Long critical section

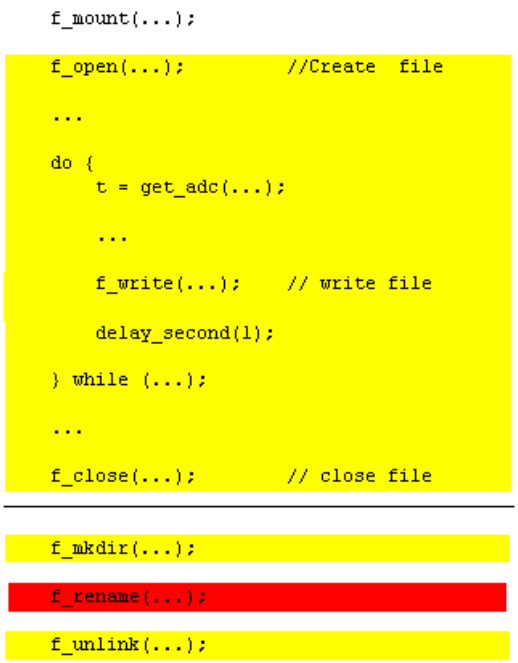
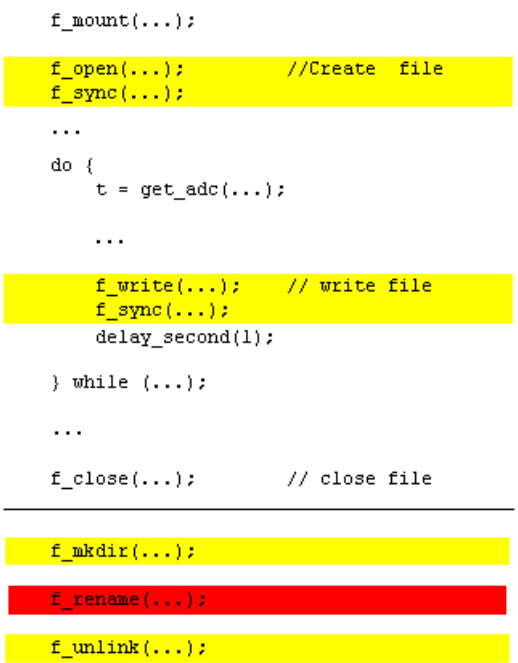


Figure 5. Minimized critical section



An interruption in the red section can cause a cross link; as a result, the object being changed can be lost. If an interruption in the yellow section is occured, there is one or more possibility listed below.

- The file data being rewrited is collapsed.
- The file being appended returns initial state.
- The file created as new is gone.
- The file created as new or overwritten remains but no content.
- Efficiency of disk use gets worse due to lost clusters.

Each case does not affect any file not opened in write mode. To minimize risk of data loss, the critical section can be minimized by minimizing the time that file is opened in write mode or using `f_sync` function as shown in Figure 5.

Various Usable Functions for FatFs Projects

These are examples of extended use of FatFs APIs. New item will be added when useful code example is found.

1. Open or Create File for Append (superseded by FA_APPEND flag added at R0.12)
2. Delete Non-empty Sub-directory (for R0.12 and later)
3. Create Contiguous File (superseded by f_expand function added at R0.12)
4. Test if the File is Contiguous or Not
5. Compatibility Checker for Storage Device Control Module
6. Performance Checker for Storage Device Control Module
7. FAT Volume Image Creator (Pre-creating built-in FAT volume)
8. Virtual Drive Feature (refer to lpc176x/ in ffsample.zip)
9. Embedded Unicode String Utilities (OEMxxx→Unicode, Unicode→OEMxxx, Unicode→Unicode)

About FatFs License

FatFs has being developped as a personal project of the author, ChaN. It is free from the code anyone else wrote at current release. Following code block shows a copy of the FatFs license document that included in the source files.

```
/*----------------------------------------------------------------------------/
/  FatFs - Generic FAT Filesystem Module  Rx.xx                               /
/-----------------------------------------------------------------------------/
/
/ Copyright (C) 20xx, ChaN, all right reserved.
/
/ FatFs module is an open source software. Redistribution and use of FatFs in
/ source and binary forms, with or without modification, are permitted provided
/ that the following condition is met:
/
/ 1. Redistributions of source code must retain the above copyright notice,
/    this condition and the following disclaimer.
/
/ This software is provided by the copyright holder and contributors "AS IS"
/ and any warranties related to this software are DISCLAIMED.
/ The copyright owner or contributors be NOT LIABLE for any damages caused
/ by use of this software.
/----------------------------------------------------------------------------*/
```

Therefore FatFs license is one of the BSD-style licenses but there is a significant feature. FatFs is mainly intended for embedded systems. In order to extend the usability for commercial products, the redistributions of FatFs in binary form, such as embedded code, binary library and any forms without source code, does not need to include about FatFs in the documentations. This is equivalent to the 1-clause BSD license. Of course FatFs is compatible with the most of open source software licenses including GNU GPL. When you redistribute the FatFs source code with any changes or create a fork, the license can also be changed to GNU GPL, BSD-style license or any open source software license that compatible with FatFs license.

Return Home