OM6681A File System User Guide

Version 1.0

Revision history

Version/Date	Modification
V1.0: 2024/01/11 Lixiaotao	- The first version

Contents

1 Introduction

1.1 Purpose and scope

The file system component is used to manage various hardware storage media, and aims to provide a unified file system interface for storage media such as internal flash, external flash, eMMC, SD cards, NAND flash, etc. It supports multiple file systems such as Fatfs, littleFS, ROFS, and abstracts the VFS (Virtual File System) on top of it to simplify the development process.

Functional features:

1) Cross-platform support: The file system SDK supports multiple storage media, including internal flash, external flash, eMMC, SD cards, NAND flash, and more.
2) Multi-file system support: The file system SDK supports multiple file systems such as Fatfs, littleFS, and ROFS. These file systems have different characteristics and advantages that can meet needs in various application scenarios. By using the SDK, it can easily switch between different file systems or use multiple file systems simultaneously to achieve optimal performance and functionality.
3) VFS abstraction layer: The file system SDK abstracts the VFS (Virtual File System) based on various file systems. VFS provides a set of common file system call interfaces and implements POSIX interfaces, such as open, read, write, and close, allowing applications to access the file system in a unified manner. By using VFS, code independent of the file system can be written, simplifying application development and maintenance.

1.2 Software architecture

The file operation component is divided into 4 layers, from bottom to top, which are the hardware driver layer, disk management layer, file system layer, and public interface layer. Different storage media rely on appropriate file systems and are abstracted into VFS interfaces to achieve the goal of unified file management.

The overall architecture is as Figure 1.2-1:

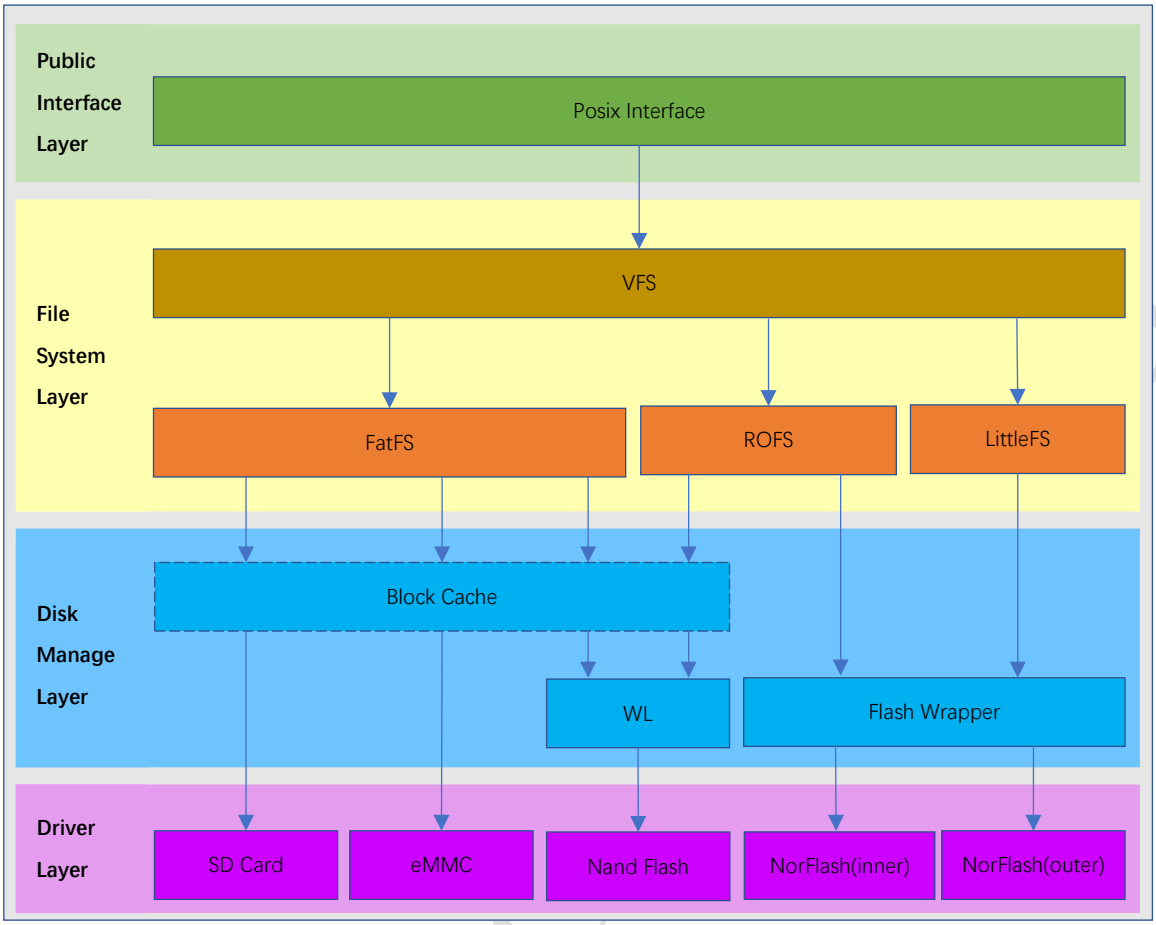

Figure 1.2-2 File System architecture

Note: Block Cache is not a mandatory component.

2 Interface description

2.1 VFS interface

2.1.1 Type decription

_MutexOps Structure: This structure defines the operations related to mutexes, if success return VFS_OK, otherwise return VFS_ERR. The parameter of all the functions are a pointer of mutex id.

mutexInit: Initializes a mutex.

mutexLock: Locks a mutex.

mutexUnlock: Unlocks a mutex.

mutexDel: Deletes a mutex.

_MemOps Structure: This structure defines the operations related to memory allocation and freeing.
memAlloc: Allocates memory.
memFree: Frees memory.

_VfsConfig Structure: This structure holds the configuration for the VFS.
mtxOps: Mutex operations.
memOps: Memory operations.
fileOpenMax: Maximum number of files that can be opened simultaneously.

_FsOps Structure: This structure defines the operations related to file system operations.
mount: Mounts a file system.
umount: Unmounts a file system.
open: Opens a file.
close: Closes a file.
read: Reads from a file.
write: Writes to a file.
lseek: Changes the file's current position.
ltell: Retrieves the current file position.
lsize: Retrieves the file size.
stat: Retrieves file status information.
truncate: Changes the file size.
unlink: Deletes a file.
rename: Renames a file or directory.
sync: Synchronizes the file system.
opendir: Opens a directory.
readdir: Reads from a directory.
closedir: Closes a directory.
mkdir: Creates a directory.
rmdir: Deletes a directory.

2.1.2 API description

Function: VFS_MutexInit
Description: Initializes a mutex.
Parameter:
mutexId: A pointer to an integer that will hold the ID of the mutex.
Return: Success returns VFS_OK, failure returns VFS_ERR.

Function: VFS_Lock
Description: Locks a mutex.
Parameter:
mutexId: The ID of the mutex to lock.
Return: Success returns VFS_OK, failure returns VFS_ERR.

Function: VFS_Unlock
Description: Unlocks a mutex.
Parameter:
mutexId: The ID of the mutex to unlock.
Return: Success returns VFS_OK, failure returns VFS_ERR.

Function: VFS_MutexDel
Description: Deletes a mutex.
Parameter:
mutexId: The ID of the mutex to delete.
Return: Success returns VFS_OK, failure returns VFS_ERR.

Function: VFS_Malloc
Description: Allocates memory.
Parameter:
size: The size of the memory block to allocate (in bytes).
Return: Success returns a pointer to the allocated memory, failure returns NULL.

Function: VFS_Free
Description: Frees memory.
Parameter:
mem: A pointer to the memory block to free.
Return: This function does not return a value.

Function: VFS_Init
Description: Initializes the VFS interface.
Parameter:
cfg: A pointer to a configuration structure.
Return: Success returns VFS_OK, failure returns VFS_ERR.

Function: VFS_DeInit
Description: Deinitializes the VFS interface.
Parameter: None.
Return: This function does not return a value.

Function: VFS_FsRegister
Description: Registers a new file system type.
Parameter:
fsType: The name of the file system type.
ops: A pointer to a structure containing file system operations.
Return: Success returns VFS_OK, failure returns VFS_ERR.

Function: VFS_FsUnregister
Description: Unregisters an existing file system type.

Parameter:

fsType: The name of the file system type to unregister.

Return: Success returns VFS_OK, failure returns VFS_ERR.

Function: VFS_Mount

Description: Mounts a file system on a target path.

Parameter:

source: The source path of the file system to mount.

target: The target path where the file system should be mounted.

fsType: The type of the file system to mount.

mntFlags: Mount flags.

data: Additional data specific to the file system type (optional).

Return: Success returns VFS_OK, failure returns VFS_ERR.

Function: VFS_Umount

Description: Unmounts a file system on a target path.

Parameter:

target: The target path where the file system should be unmounted.

Return: Success returns VFS_OK, failure returns VFS_ERR.

Function: VFS_Open

Description: Opens a file.

Parameter:

path: The path of the file to open.

flags: Flags to specify the mode of opening the file (e.g., read, write, append).

Return: Success returns a file descriptor (non-negative integer), failure returns VFS_ERR.

Function: VFS_Close

Description: Closes a file.

Parameter:

fd: The file descriptor of the file to close.

Return: Success returns VFS_OK, failure returns VFS_ERR.

Function: VFS_Read

Description: Reads data from a file.

Parameter:

fd: The file descriptor of the file to read from.

buff: A buffer to store the read data.

size: The size of the buffer in bytes.

Return: Success returns the number of bytes read, failure returns VFS_ERR.

Function: VFS_Write

Description: Writes data to a file.

Parameter:

fd: The file descriptor of the file to write to.
buff: A pointer to the data to write.
size: The size of the data in bytes.
Return: Success returns the number of bytes written, failure returns VFS_ERR.

Function: VFS_Lseek
Description: Sets the current file offset.
Parameter:
fd: The file descriptor of the file to seek in.
off: The offset to seek to, relative to whence.
whence: An integer representing the reference point for the offset (e.g., SEEK_SET, SEEK_CUR, SEEK_END).
Return: Success returns the new file offset, failure returns VFS_ERR.

Function: VFS_Ltell
Description: Gets the current file offset.
Parameter:
fd: The file descriptor of the file to get the offset of.
Return: Success returns the current file offset, failure returns VFS_ERR.

Function: VFS_Lsize
Description: Gets the size of a file.
Parameter:
fd: The file descriptor of the file to get the size of.
Return: Success returns the size of the file in bytes, failure returns VFS_ERR.

Function: VFS_Truncate
Description: Truncates a file to a specific size.
Parameter:
fd: The file descriptor of the file to truncate.
size: The new size of the file in bytes.
Return: Success returns VFS_OK, failure returns VFS_ERR.

Function: VFS_Sync
Description: Synchronizes a file's in-memory state with its storage device.
Parameter:
fd: The file descriptor of the file to synchronize.
Return: Success returns VFS_OK, failure returns VFS_ERR.

Function: VFS_Stat
Description: Gets file status information.
Parameter:
path: The path of the file to get status information for.
stat: A pointer to a struct stat to store the status information.

Return: Success returns VFS_OK, failure returns VFS_ERR.

Function: VFS_Unlink
Description: Deletes a file.
Parameter:
path: The path of the file to delete.
Return: Success returns VFS_OK, failure returns VFS_ERR.

Function: VFS_Rename
Description: Renames a file or directory.
Parameter:
oldPath: The current path of the file or directory.
newPath: The new path of the file or directory.
Return: Success returns VFS_OK, failure returns VFS_ERR.

Function: VFS_Opendir
Description: Opens a directory for reading.
Parameter:
path: The path of the directory to open.
Return: Success returns a DIR pointer, failure returns NULL.

Function: VFS_Readdir
Description: Reads the next entry from a directory.
Parameter:
pd: A DIR pointer to the directory to read from.
Return: Success returns a pointer to a dirent structure, failure returns NULL.

Function: VFS_Closedir
Description: Closes a directory.
Parameter:
pd: A DIR pointer to the directory to close.
Return: Success returns VFS_OK, failure returns VFS_ERR.

Function: VFS_Mkdir
Description: Creates a new directory.
Parameter:
path: The path of the new directory.
mode: The permissions for the new directory.
Return: Success returns VFS_OK, failure returns VFS_ERR.

Function: VFS_Rmdir
Description: Deletes an empty directory.
Parameter:
path: The path of the directory to delete.

Return: Success returns VFS_OK, failure returns VFS_ERR.

2.2 FatFS interface

Please refer to the official documentation of FatFS.

2.3 LittleFS interface

Please refer to the official documentation of LittleFS.

2.4 ROFS interface

ROFS is a simple and efficient file system that supports nand flash and nor flash. Compared to other file systems, it has high efficiency in accessing disks but does not have particularly rich features, only supporting access to the first-level directory (root directory).

Function: RO_Init
Description: Initializes the RO file system.
Parameter: None.
Return: returns RO_ERR_OK if successful, an error code otherwise.

Function: RO_Uninit
Description: Uninitializes the RO file system.
Parameter: None.
Return: returns RO_ERR_OK if successful, an error code otherwise.

Function: RO_Mount
Description: Mounts a file system partition.
Parameter:
mntCfg: Mount configuration structure.
Return: returns RO_ERR_OK if successful, an error code otherwise.

Function: RO_Unmount
Description: Unmounts a file system partition.
Parameter:
mntCfg: Mount configuration structure.
Return: returns RO_ERR_OK if successful, an error code otherwise.

Function: RO_Open
Description: Opens a file for reading or writing.
Parameter:

mntCfg: Mount configuration structure.
file: File descriptor structure.
path: Path of the file to open.
oflags: Open flags (e.g., RO_O_RDONLY for read-only).
Return: returns RO_ERR_OK if successful, an error code otherwise.

Function: RO_Close
Description: Closes a file opened with RO_Open.
Parameter:
mntCfg: Mount configuration structure.
file: File descriptor structure.
Return: returns RO_ERR_OK if successful, an error code otherwise.

Function: RO_Read
Description: Reads data from a file opened with RO_Open.
Parameter:
mntCfg: Mount configuration structure.
file: File descriptor structure.
buffer: Buffer to store the read data.
len: Number of bytes to read.
Return: Number of bytes read, -1 on error.

Function: RO_Write
Description: Writes data to a file opened with RO_Open.
Parameter:
mntCfg: Mount configuration structure.
Return: Number of bytes writen, -1 on error.

Function: RO_Sync
Description: Synchronizes the file system to the storage device.
Parameter:
mntCfg: Mount configuration structure.
file: File descriptor structure.
Return: returns RO_ERR_OK if successful, an error code otherwise.

Function: RO_Seek
Description: Sets the file position to the specified offset.
Parameter:
mntCfg: Mount configuration structure.
file: File descriptor structure.
offset: Offset in bytes from the beginning of the file.
Return: New file position if successful, -1 on error.

Function: RO_Tell

Description: Retrieves the current file position.
Parameter:
mntCfg: Mount configuration structure.
file: File descriptor structure.
Return: Current file position if successful, -1 on error.

Function: RO_Size
Description: Retrieves the size of a file.
Parameter:
mntCfg: Mount configuration structure.
file: File descriptor structure.
Return: File size in bytes if successful, -1 on error.

Function: RO_Remove
Description: Removes a file or directory.
Parameter:
mntCfg: Mount configuration structure.
path: Path of the file or directory to remove.
Return: returns RO_ERR_OK if successful, an error code otherwise.

Function: RO_Addr2Path
Description: Converts an address to a path.
Parameter:
addr: Address to convert.
pathBuff: Buffer to store the path.
pathBuffLen: Length of the path buffer.
Return: returns RO_ERR_OK if successful, an error code otherwise.

Function: RO_DiskInit
Description: Initializes the disk device. This function must be called before any other disk operations can be performed.
Parameter:
mntCfg: Mount configuration structure.
DiskCfg: Disk configuration structure containing parameters for initializing the disk device.
Return: returns RO_ERR_OK if successful, an error code otherwise.

2.5 WL interface

WL is a component used to manage nand flash. It can map nand flash pages to WL sectors, and supports functions such as wear leveling, bad block management, and unexpected power loss for nand flash.

2.5.1 Macro Description

The following macros are used in the WL component to define error codes and other relevant values.

Macro: WL_OK

Description: This macro represents a successful operation. It is typically used as a return code to indicate that an operation completed without any errors.

Macro: WL_ERROR

Description: This macro represents a generic error. It is used when an operation fails for reasons other than those specifically accounted for by other error codes.

Macro: WL_NO_SECTORS

Description: This macro indicates that there are no sectors available for reading or writing.

Macro: WL_SECTOR_NOT_FOUND

Description: This macro indicates that the requested sector could not be found. It may be returned if the specified sector number is out of range or if the sector has been deleted or corrupted.

Macro: WL_NO_PAGES

Description: This macro indicates that there are no pages available for writing. It is typically returned when all pages in the NAND device are already full.

Macro: WL_NAND_ERROR_CORRECTED

Description: This macro indicates that a NAND error was corrected during a read or write operation. It provides feedback that the system is capable of handling and correcting errors that may occur during NAND operations.

Macro: WL_NAND_ERROR_NOT_CORRECTED

Description: This macro indicates that a NAND error could not be corrected during a read or write operation. It indicates that the system encountered an uncorrectable error and may require intervention or further investigation.

Macro: WL_NO_MEMORY

Description: This macro indicates that there is insufficient memory available to complete the requested operation. It may be returned if the system lacks the necessary resources to process the request.

Macro: WL_INVALID_BLOCK

Description: This macro indicates that the specified block is invalid or corrupted. It may be returned if the block number provided is out of range or if the block has been marked as bad or unusable.

Macro: WL_INVALID_FORMAT

Description: This macro indicates that the requested operation cannot be performed because the storage medium's format is invalid or unsupported. It may be returned if the medium is not formatted correctly or if the format is not recognized by the system.

Macro: WL_INVALID_SECTOR_MAP

Description: This macro indicates that the sector map is invalid or corrupted. It may be returned if the

sector map data is not consistent or if it has been corrupted during operation.

Macro: WL_INIT_FAILED
Description: This macro indicates that the initialization process for the Wear Leveling component has failed. It may be returned if the necessary resources are not available or if the initialization steps cannot be completed successfully.

Macro: WL_ALLOCATION_FAILED
Description: This macro indicates that memory allocation has failed during operation. It may be returned if there is insufficient memory available to complete the requested operation or if memory allocation fails unexpectedly.

Macro: WL_MUTEX_CREATE_FAILED
Description: This macro indicates that the creation of a mutex (mutual exclusion object) has failed. It may be returned if the system is unable to create a mutex to synchronize access to shared resources.

Macro: WL_DIVER_INIT_FAILED
Description: This macro indicates that the initialization process for the diver component has failed. It may be returned if the necessary resources are not available or if the initialization steps cannot be completed successfully.

Macro: WL_DIVER_READ_FAILED
Description: This macro indicates that a read operation has failed. It may be returned if there is a problem with the storage medium, the data is corrupted, or another read-related error occurs.

Macro: WL_DIVER_WRITE_FAILED
Description: This macro indicates that a write operation has failed. It may be returned if there is a problem with the storage medium, if the data is corrupted, or another write-related error occurs.

Macro: WL_DIVER_ERASE_FAILED
Description: This macro indicates that an erase operation has failed. It may be returned if there is a problem with the storage medium, if the erase process cannot be completed successfully, or another erase-related error occurs.

Macro: WL_DIVER_ERASE_VERIFY_FAILED
Description: This macro indicates that the verification of an erase operation has failed. It may be returned if the erase process was not completed successfully or if there is a problem verifying the erase process.

Macro: WL_DIVER_GET_BLOCK_STATUS_FAILED
Description: This macro indicates that getting the block status has failed. It may be returned if there is a problem determining the status of a block or if another error occurs related to block status retrieval.

Macro: WL_DIVER_SET_BLOCK_STATUS_FAILED
Description: This macro indicates that setting the block status has failed. It may be returned if there is a

problem modifying the status of a block or if another error occurs related to block status modification.

Macro: WL_DIVER_GET_EXTRA_BYTES_FAILED
Description: This macro indicates that getting extra bytes has failed. It may be returned if there is a problem accessing or retrieving the extra bytes or if another error occurs related to extra bytes retrieval.

Macro: WL_DIVER_SET_EXTRA_BYTES_FAILED
Description: This macro indicates that setting extra bytes has failed. It may be returned if there is a problem modifying or storing the extra bytes or if another error occurs related to extra bytes modification.

2.5.2 Structure description

Data structure: WlNandCfg_t
The WlNandCfg_t structure defines the configuration parameters for WL component. It is used for WL_Open function, so implement these configurations before call WL_Open. It contains the following fields:

blockOffset: The block offset within the NAND flash device.
blockCount: The number of blocks to be managed by the wear-leveling component.
pagesPerBlock: The number of pages per block in the NAND flash device.
bytesPerPage: The number of bytes per page in the NAND flash device.
pageBuffer: A buffer to store data read from or written to the NAND flash pages.
mapBuffer: A buffer used for mapping logical sectors to physical pages. if direct map enabled, the size should be 4*pagesPerBlock*blockCount bytes at least, otherwise, the size should be 8*16 bytes at least.
mapBufferWords: The size of the mapBuffer in words (32-bit).
directMapEnable: A flag indicating whether to enable direct mapping of logical blocks to physical blocks. See the description of mapBuffer for details.
driverInit: A function pointer to the driver's initialization function.
driverSystemError: A function pointer to the driver's system error handling function.
driverRead: A function pointer to the driver's read function.
driverWrite: A function pointer to the driver's write function.
driverBlockErase: A function pointer to the driver's block erase function.
driverBlockErasedVerify: A function pointer to the driver's block erase verification function.
driverPageErasedVerify: A function pointer to the driver's page erase verification function.
driverBlockStatusGet: A function pointer to the driver's block status getting function.
driverBlockStatusSet: A function pointer to the driver's block status setting function.
driverExtraDataGet: A function pointer to the driver's extra data getting function.
driverExtraDataSet: A function pointer to the driver's extra data setting function.

2.5.3 API Description

Function: WL_Init
Description: Initializes the wear leveling system.
Parameter:
sysCfg: System configuration structure.

Return: Success status (WL_OK for success).

Function: WL_Open
Description: Opens a NAND device partition for wear leveling operations.
Parameter:
cfg: NAND device configuration structure.
Return: Pointer to the opened NAND device structure.

Function: WL_GetDevCfg
Description: Retrieves the configuration of a NAND device.
Parameter:
nandDev: NAND device structure.
Return: Pointer to the retrieved device configuration structure.

Function: WL_GetTotalSectorNumber
Description: Retrieves the total number of sectors in a NAND device.
Parameter:
nandDev: NAND device structure.
number: Pointer to store the total number of sectors.
Return: Success status (WL_OK for success).

Function: WL_Close
Description: Closes an opened NAND device.
Parameter:
nandDev: NAND device structure.
Return: Success status (WL_OK for success).

Function: WL_Read
Description: Reads data from a logical sector of a NAND device into a buffer.
Parameter:
nandDev: NAND device structure.
logicalSector: Logical sector number to read.
buffer: Buffer to store the read data.
Return: Success status (WL_OK for success).

Function: WL_Write
Description: Writes data from a buffer to a logical sector of a NAND device.
Parameter:
nandDev: NAND device structure.
logicalSector: Logical sector number to write.
buffer: Buffer containing the data to write.
Return: Success status (WL_OK for success).

Function: WL_Release

Description: Releases a logical sector from a NAND device.
Parameter:
nandDev: NAND device structure.
logicalSector: Logical sector number to release.
Return: Success status (WL_OK for success).

Function: WL_ExtCacheEnable
Description: Enables an external cache for a NAND device, such as block status cache, it will improve the WL performance, but more memory needed.
Parameter:
nandDev: NAND device structure.
memory: Pointer to the memory for the cache.
size: Size of the cache memory.
Return: Success status (WL_OK for success).

Function: WL_ExtCacheDisable
Description: Disables the external cache of a NAND device, only page map cache used.
Parameter:
nandDev: NAND device structure.
Return: Success status (WL_OK for success).

Function: WL_PartialDefragment
Description: Performs partial defragmentation on a NAND device.
Parameter:
nandDev: NAND device structure.
max_blocks: Maximum number of blocks to defragment.
Return: Success status (WL_OK for success).

Function: WL_Defragment
Description: Performs defragmentation on a NAND device.
Parameter:
nandDev: NAND device structure.
Return: Success status (WL_OK for success).

2.6 Block Cache

2.6.1 Structure description

Data structure: blk_cache_cfg_t
The blk_cache_cfg_t structure defines the configuration parameters for a block cache component of a block device. It contains the following fields:
sector_size: The size of each sector in bytes.
cache_depth: The depth of the cache.

cache_mem_type: An enumerated type om_mem_type_t representing the type of memory used for the cache.

2.6.2 API description

Function: blk_cache_open
Description: This function opens a block cache using the provided configuration.
Parameter:
cfg: Pointer to the block cache configuration structure containing the necessary parameters for cache initialization.
Return:
Pointer to the block cache object that can be used for further operations.

Function: blk_cache_close
Description: This function closes the block cache and releases any associated resources.
Parameter:
bc: Pointer to the block cache object to be closed.
Return: None.

Function: blk_cache_read
Description: This function reads a block from the cache. If the block is not present in the cache, it may be read from the underlying device and added to the cache by blk_cache_update function.
Parameter:
bc: Pointer to the block cache object.
sector: The sector number of the block to be read.
buffer: A buffer to store the read block.
Return: An value indicating the result of the operation. It can be one of the following values:
BLK_CACHE_OK: The operation was successful.
BLK_CACHE_NOT_CACHED: The requested block was not found in the cache.
BLK_CACHE_ERROR: An error occurred during the operation.

Function: blk_cache_update
Description: This function updates a block in the cache. If the block is not present in the cache, it may be read from the underlying device and added to the cache. Any oldest existing cached data for that sector will be replaced with the new data.
Parameter:
bc: Pointer to the block cache object.
sector: The sector number of the block to be updated.
buffer: A buffer containing the new data for the block.
Return: An value indicating the result of the operation. It can be one of the following values:
BLK_CACHE_OK: The operation was successful.
BLK_CACHE_ERROR: An error occurred during the operation.

3 Application

3.1 VFS application

3.1.1 Project configuration

If VFS is needed，modify menuconfig as follows to enable VFS:

Components -> Filesystem -> Enable Virtual filesystem (VFS)

3.1.2 Porting

VFS has been adapted to some file systems, such as FatFS, ROFS and LittleFS. If you need to adapt to other file systems, you can follow the steps below for porting.

1) Implement function interfaces for VFS configuration, then initialize VFS by VFS_Init, the parameter is the configuration, and it only needs to be initialized once.
2) Implement function interfaces for VFS ops, then register it to VFS by VFS_FsRegister.

After the above steps, you can use mount, open, read, close, and other posix interfaces to mount the file system and manipulate files. It should be noted that the mount path must be unix style. If you will not use VFS, you need unregister the VFS and uninitialized it by VFS_Deinit and VFS_Unregister.

3.1.3 FS initialize

The steps for initializing the file system are as follows:

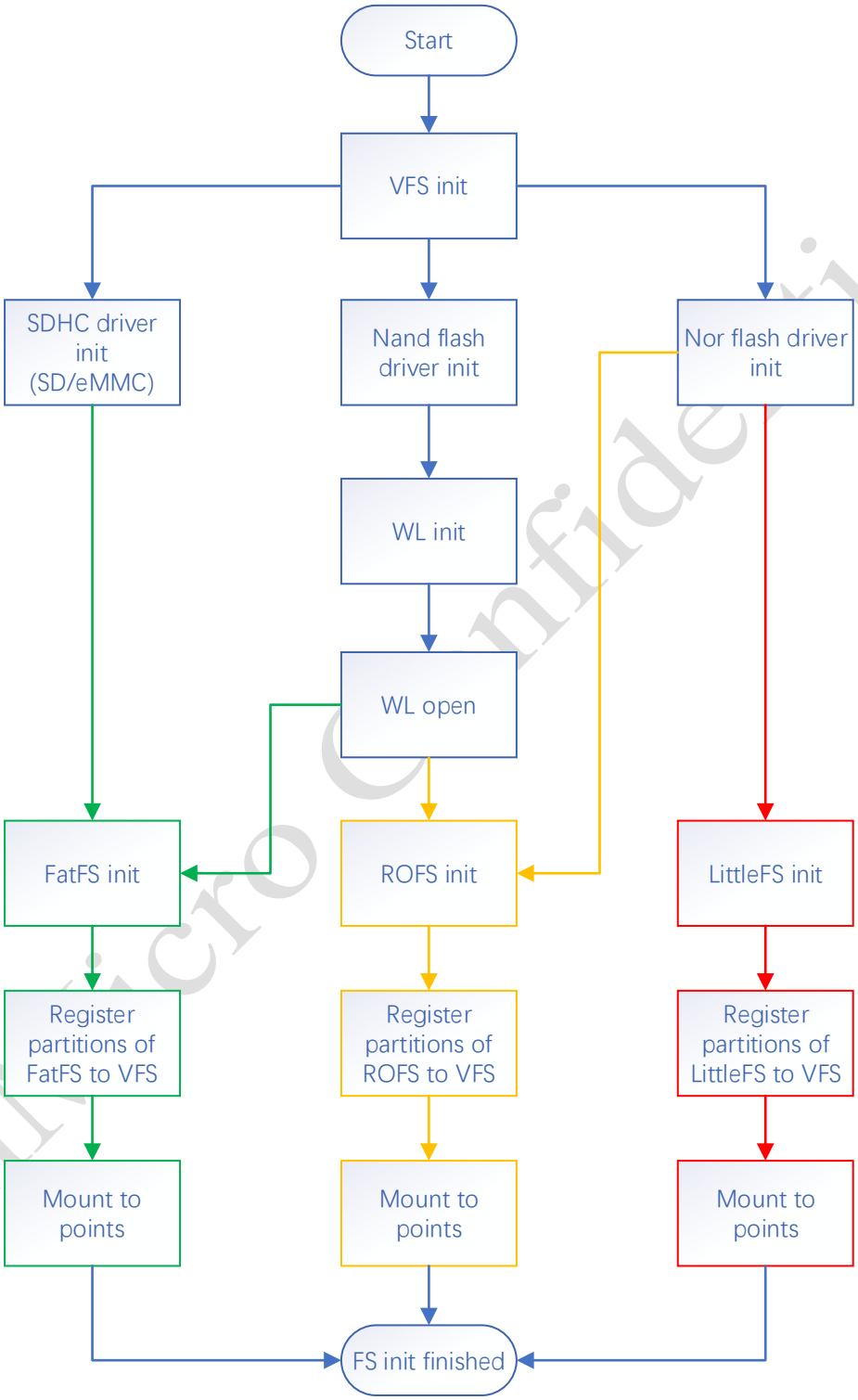

Figure 3.1-1 FS init

Note: SD card, eMMC, and nand flash can only exist at the same time.

3.2 FatFS application

FatFS can be used to SD card/eMMC and nand flash, Only one can exist simultaneously.

3.2.1 FatFS for SD/eMMC

If SD card or eMMC is used, modify the project as follows:

1) Modify **RTE_om6681a_core0.h** of the project, ensure that the configuration of the following item is correct:

```
#define RTE_SDHC                    (1)
```

2) Modify menuconfig for project:

Components -> Filesystem -> Generic FAT/exFAT filesystem -> disk type -> （SD or eMMC）

3.2.2 FatFS for nand flash

If nand flash is used, modify the project as follows:

1) Modify **RTE_om6681a_core0.h** of the project, ensure that the configuration of the following 3 items is correct:

```
#define RTE_OSPI1_PSRAM          (0 & RTE_OSPI1)
#define RTE_OSPI1_FLASH          (0 & RTE_OSPI1)
#define RTE_OSPI1_NAND_FLASH     (1 & RTE_OSPI1)
```

2) Modify menuconfig for the project:

Components -> Filesystem -> Generic FAT/exFAT filesystem -> disk type -> Nand Flash

3.3 LittleFS application

It has already adapted to inner flash and outer flash, modify menuconfig as follows to enable it:

Components -> Filesystem -> A little fail-safe filesystem

If the parameters is not appropriate, modify them.

3.4 ROFS application

ROFS can be used to nand flash and nor flash.

3.4.1 ROFS for nand flash

To enable nand flash support, modify the project as follows:

1) Modify **RTE_om6681a_core0.h** of the project, ensure that the configuration of the following 3 items is correct:

```
1.  #define RTE_OSPI1_PSRAM          (0 & RTE_OSPI1)
2.  #define RTE_OSPI1_FLASH          (0 & RTE_OSPI1)
3.  #define RTE_OSPI1_NAND_FLASH     (1 & RTE_OSPI1)
```

2) Modify menuconfig as follows:

Components -> Filesystem -> A read only filesystem (ROFS) based on nand/nor flash -> enable ROFS for nand flash

3.4.2 ROFS for nor flash

To enable nor flash support, modify the project as follows:

1) If using outer flash, modify **RTE_om6681a_core0.h** of the project, ensure that the configuration of the following 3 items is correct:

```
1.  #define RTE_OSPI1_PSRAM          (0 & RTE_OSPI1)
2.  #define RTE_OSPI1_FLASH          (1 & RTE_OSPI1)
3.  #define RTE_OSPI1_NAND_FLASH     (0 & RTE_OSPI1)
```

2) Modify menuconfig as follows:

Components -> Filesystem -> A read only filesystem (ROFS) based on nand/nor flash -> enable ROFS for nor flash

3.5 WL application

WL component is used to manage nand flash, when FatFS or ROFS is based on nand flash, WL should be enabled.

1) Modify **RTE_om6681a_core0.h** of the project, ensure that the configuration of the following 3 items is correct:

```
1.  #define RTE_OSPI1_PSRAM          (0 & RTE_OSPI1)
2.  #define RTE_OSPI1_FLASH          (0 & RTE_OSPI1)
3.  #define RTE_OSPI1_NAND_FLASH     (1 & RTE_OSPI1)
```

2) Modify menuconfig as follows:

Components -> Filesystem -> Nand flash WL component

If WL log is enabled, it will output log when any error occurred in WL driver layer.

If the map buffer of WL is lost, the WL object of corresponding partition shoud be closed and reopen by WL_Close and WL_Open.

3.6 Block cache application

Block cache is used for Improving the access speed of block devices, it is adapted to nand flash now. It requires some ram when enabled. Enabling block cache component needs Modify menuconfig as follows:

Components -> Filesystem -> Block device access with cach