



**Hi3861 V100 / Hi3861L V100 SPIFFS**

## **User Guide**

<b>Issue</b>	<b>01</b>
<b>Date</b>	<b>2020-04-30</b>

**Copyright © HiSilicon (Shanghai) Technologies Co., Ltd. 2020. All rights reserved.**

No part of this document may be reproduced or transmitted in any form or by any means without prior written consent of HiSilicon (Shanghai) Technologies Co., Ltd.

## **Trademarks and Permissions**



**HISILICON**, and other HiSilicon icons are trademarks of HiSilicon Technologies Co., Ltd.

All other trademarks and trade names mentioned in this document are the property of their respective holders.

## **Notice**

The purchased products, services and features are stipulated by the contract made between HiSilicon and the customer. All or part of the products, services and features described in this document may not be within the purchase scope or the usage scope. Unless otherwise specified in the contract, all statements, information, and recommendations in this document are provided “AS IS” without warranties, guarantees or representations of any kind, either express or implied.

The information in this document is subject to change without notice. Every effort has been made in the preparation of this document to ensure accuracy of the contents, but all statements, information, and recommendations in this document do not constitute a warranty of any kind, express or implied.

## **HiSilicon (Shanghai) Technologies Co., Ltd.**

Address: New R&D Center, 49 Wuhe Road,  
Bantian, Longgang District,  
Shenzhen 518129 P. R. China

Website: <https://www.hisilicon.com/en/>

Email: [support@hisilicon.com](mailto:support@hisilicon.com)



# About This Document

## Purpose

This document describes the file system development of Hi3861 V100 and Hi3861L V100, including the interface implementation mechanism and usage description.

## Related Versions

The following table lists the product versions related to this document.

Product Name	Version
Hi3861	V100
Hi3861L	V100



## Intended Audience

The document is intended for:



- Technical support engineers
- Software development engineers

## Symbol Conventions

The symbols that may be found in this document are defined as follows.

Symbol	Description
	Indicates a hazard with a high level of risk which, if not avoided, will result in death or serious injury.
	Indicates a hazard with a medium level of risk which, if not avoided, could result in death or serious injury.



Symbol	Description
 CAUTION	Indicates a hazard with a low level of risk which, if not avoided, could result in minor or moderate injury.
NOTICE	Indicates a potentially hazardous situation which, if not avoided, could result in equipment damage, data loss, performance deterioration, or unanticipated results. NOTICE is used to address practices not related to personal injury.
 NOTE	Supplements the important information in the main text. NOTE is used to address information not related to personal injury, equipment damage, and environment deterioration.

## Change History

Issue	Date	Change Description
01	2020-04-30	This issue is the first official release.
00B02	2020-02-12	<ul style="list-style-type: none"><li>In <a href="#">2.1.1 Overview</a>, the NOTICE about file system operations is added.</li><li>In <a href="#">2.1.2 Development Process</a>, the description of the interface for obtaining error codes is added to <a href="#">Table 2-1</a>.</li></ul>
00B01	2020-01-15	This issue is the first draft release.



# Contents

About This Document.....	i
1 Introduction.....	1
2 System APIs.....	2
2.1 SPIFFS.....	2
2.1.1 Overview.....	2
2.1.2 Development Process.....	2
2.1.3 Precautions.....	4
2.1.4 Programming Samples.....	4



# 1 Introduction

---

The SPI flash file system (SPIFFS) is designed for small embedded systems. It allows users to open, close, read, write, and delete data on the NOR flash by using APIs.



# 2 System APIs

## 2.1 SPIFFS

## 2.1 SPIFFS

### 2.1.1 Overview

SPIFFS is a file system intended for SPI NOR flash devices on embedded targets. It supports wear leveling, file system consistency checks, and more.

#### NOTICE

Compared with direct operations on flash partitions, file system operations increase the number of flash erase times and shorten the flash service life. Therefore, you are advised not to use the file system in scenarios where the flash memory is frequently read and written. Instead, you are advised to customize the flash partitions and directly operate the flash memory.

### 2.1.2 Development Process

#### Application Scenario

SPIFFS is a file system built for small embedded systems.

- Designed for low-RAM usage scenarios of micro-controllers
- With APIs to open, close, read, write, delete, and enumerate files
- Implementing static wear leveling
- Supporting system consistency check

#### Functions

[Table 2-1](#) describes the APIs provided by the SPIFFS.



**Table 2-1** SPIFFS interfaces

Function	API Name	Description
File system initialization	hi_fs_init	Initializes dynamic parameters of the file system and mounts the file system (initialized by default).
File I/O operations	hi_open	Opens or creates a file. <ul style="list-style-type: none"><li>• <b>HI_FS_O_RDONLY</b>: Opens a file in read-only mode.</li><li>• <b>HI_FS_O_WRONLY</b>: Opens a file in write-only mode.</li><li>• <b>HI_FS_O_RDWR</b>: Opens a file in read/write mode.</li><li>• <b>HI_FS_O_CREAT</b>: Creates a file if the file to be opened does not exist.</li><li>• <b>HI_FS_O_EXCL</b>: Checks whether a file exists if <b>HI_FS_O_CREAT</b> is enabled. If not, create a file. Otherwise, an error occurs when you open the file.</li><li>• <b>HI_FS_O_TRUNC</b>: If a file exists and is opened in write mode, this command clears the file length and the data stored in the file disappears.</li><li>• <b>HI_FS_O_APPEND</b>: Locates the end of a file, and appends data to the end.</li></ul>
	hi_close	Closes a file handle. Pending write operations should be complete before closing the handle.
	hi_read	Reads a file and returns the read bytes. Returns -1 and set an error code if this read operation fails. If the end of the file has been reached before this interface is called, 0 is returned.
	hi_write	Writes data to a file. If the operation succeeds, the number of written bytes is returned. Otherwise, -1 is returned and an error code is set.
	hi_unlink	Deletes a file according to the path name.
Read/write file offset setting	hi_lseek	Sets the file read/write offset. <b>HI_FS_SEEK_SET</b> : Shifts <b>offset</b> bytes from the file header. <b>HI_FS_SEEK_CUR</b> : Shifts <b>offset</b> bytes from the position of the current read/write pointer. <b>HI_FS_SEEK_END</b> : Sets the offset to <b>file size+offset</b> bytes. <b>offset</b> must be a negative value.
Enumerating files	hi_enum_file	Enumerate all files in the <b>root</b> directory.





Function	API Name	Description
Obtaining the file size	hi_stat	Obtains the file size according to the file name.
Obtaining error codes	hi_get_fs_error	Obtains the error codes of file system interface calling failures. You can search for an error code of the file system in <b>hi_errno.h</b> .

### 2.1.3 Precautions

- The SPIFFS does not support multi-level directories.
- A file name cannot exceed 31 bytes (excluding the end character).
- Up to 32 files can be opened at the same time.
- If 32 files are opened, deleting files is not allowed. You need to call the **hi\_close** interface to close one of the opened files, then you can delete files.

### 2.1.4 Programming Samples

#### Reading/Writing Files

The programming sample is as follows:

```
hi_void example_fs_rw(hi_void)
{
    char buf[12]={0};
    /* create a file, delete previous if it already exists, and open it for reading and writing */
    hi_s32 fd = hi_open("my_file", HI_FS_O_CREAT | HI_FS_O_TRUNC | HI_FS_O_RDWR);
    if (fd < 0) {
        printf("errno 0x%x\n", hi_get_fs_error());
        return;
    }
    // write to it
    if (hi_write(fd, "Hello world",12) < 0) {
        printf("errno 0x%x\n", hi_get_fs_error());
        return;
    }
    // close it
    if (hi_close(fd) < 0) {
        printf("errno 0x%x\n", hi_get_fs_error());
        return;
    }
    // open it
    fd = hi_open("my_file", HI_FS_O_RDWR);
    if (fd < 0) {
        printf("errno 0x%x\n", hi_get_fs_error());
        return;
    }
    // read it
    if (hi_read(fd, (hi_char *)buf, sizeof(buf)) < 0) {
```



```
        printf("errno 0x%x\n", hi_get_fs_error());  
        return;  
    }  
    // close it  
    if (hi_close(fd) < 0) {  
        printf("errno 0x%x\n", hi_get_fs_error());  
        return;  
    }  
    // check it  
    printf("--> %s <--\n", buf);  
}
```

Verification

```
--> Hello world <--
```

## Setting the Read/Write Offset

The programming sample is as follows:

```
hi_void example_fs_lseek(hi_void)  
{  
    u8_t buf[128];  
    int i, res;  
    u8_t test;  
    for (i = 0; i < 128; i++)  
    {  
        buf[i] = i;  
    }  
    // create a file with some data in it  
    hi_s32 fd = hi_open("somedata", HI_FS_O_CREAT | HI_FS_O_RDWR);  
    if (fd < 0)  
    {  
        printf("errno 0x%x\n", hi_get_fs_error());  
        return;  
    }  
    res = hi_write(fd, (hi_char*)buf, 128);  
    if (res < 0)  
    {  
        printf("errno 0x%x\n", hi_get_fs_error());  
        return;  
    }  
    res = hi_close(fd);  
    if (res < 0)  
    {  
        printf("errno 0x%x\n", hi_get_fs_error());  
        return;  
    }  
    // open for reading  
    fd = hi_open("somedata", HI_FS_O_CREAT | HI_FS_O_RDONLY);  
    if (fd < 0)  
    {  
        printf("errno 0x%x\n", hi_get_fs_error());  
        return;  
    }  
    // read the last byte of the file  
    res = hi_lseek(fd, -1, HI_FS_SEEK_END);  
    if (res < 0)  
    {  
        printf("errno 0x%x\n", hi_get_fs_error());  
        return;  
    }  
}
```



```
res = hi_read(fd, (hi_char*)&test, 1);
if (res < 0)
{
    printf("errno 0x%x\n", hi_get_fs_error());
    return;
}
printf("last byte:%i\n", test); // prints "last byte:127"
// read the middle byte of the file
res = hi_lseek(fd, 64, HI_FS_SEEK_SET);
if (res < 0)
{
    printf("errno 0x%x\n", hi_get_fs_error());
    return;
}
res = hi_read(fd, (hi_char*)&test, 1);
if (res < 0)
{
    printf("errno 0x%x\n", hi_get_fs_error());
    return;
}
printf("middle byte:%i\n", test); // prints "middle byte:64"
// skip 3 bytes from current offset and read next byte (NB we read one byte previously also)
res = hi_lseek(fd, 3, HI_FS_SEEK_CUR);
if (res < 0)
{
    printf("errno 0x%x\n", hi_get_fs_error());
    return;
}
res = hi_read(fd, (hi_char*)&test, 1);
if (res < 0)
{
    printf("errno 0x%x\n", hi_get_fs_error());
    return;
}
printf("middle+4 byte:%i\n", test); // prints "middle+4 byte:68"
res = hi_close(fd);
if (res < 0)
{
    printf("11errno 0x%x\n", hi_get_fs_error());
    return;
}
}
```

#### Verification

last byte:127  
middle byte:64  
middle+4 byte:68

## Deleting All Files

The programming sample is as follows:

```
void example_fs_remove_file(hi_void)
{
    char* buf = NULL;
    //After all files are successfully enumerated, the buffer must be released. Otherwise,
    memory leakage occurs.
    int ret = hi_enum_file("/", &buf);
    if (ret < 0)
    {
        printf("11errno 0x%x\n", hi_get_fs_error());
    }
}
```



```
        return;
    }
    file_list* file = (file_list*)buf;
    do
    {
        int res = hi_unlink(file->name);
        if (res < 0) {
            printf("errno 0x%x\n", hi_get_fs_error());
            break;
        }
        else
        {
            printf("delete %s success.\n", file->name);
        }
        if (!file->next_offset)
        {
            break;
        }
        file = (file_list*)((char*)file + file->next_offset);
    }while(TRUE);
    hi_free(0, buf);
}
```

#### Verification

Delete my\_file success.  
Delete somedata success.

## Enumerating All Files in the root Directory.

The programming sample is as follows:

```
hi_void example_fs_enum_file(hi_void)
{
    int ret = 0;
    char* buf = NULL;
    ret = hi_enum_file("/", &buf);
    if (ret < 0) {
        printf("1errno 0x%x\n", hi_get_fs_error());
        return;
    }
    file_list* file = (file_list*)buf;
    do
    {
        printf("%s, size=0x%x\n", file->name, file->size);
        if (!file->next_offset)
        {
            break;
        }
        file = (file_list*)((char*)file + file->next_offset);
    }while(TRUE);
    hi_free(0, buf);
}
```

#### NOTE

- If no file exists in the root directory, the **HI\_ERR\_FS\_NO\_MORE\_FILES** error is returned.
- If the offset of the next file is **0**, the current file is the last file.
- The returned buffer must be freed by the user to avoid memory leaks.

**Table 2-2** shows the format of files stored in the buffer.



Table 2-2 File storage format

File size	Offset position of the next file	File name	File size	Offset position of the next file	File name	.....	File size	Zero offset of the next file	File name
-----------	----------------------------------	-----------	-----------	----------------------------------	-----------	-------	-----------	------------------------------	-----------