Embedded in Life

An IXYS Company

eZ80® CPU

Zilog File System

Reference Manual

RM003914-1211

www.zilog.com

Warning: DO NOT USE THIS PRODUCT IN LIFE SUPPORT SYSTEMS.

LIFE SUPPORT POLICY

ZILOG'S PRODUCTS ARE NOT AUTHORIZED FOR USE AS CRITICAL COMPONENTS IN LIFE SUPPORT DEVICES OR SYSTEMS WITHOUT THE EXPRESS PRIOR WRITTEN APPROVAL OF THE PRESIDENT AND GENERAL COUNSEL OF ZILOG CORPORATION.

As used herein

Life support devices or systems are devices which (a) are intended for surgical implant into the body, or (b) support or sustain life and whose failure to perform when properly used in accordance with instructions for use provided in the labeling can be reasonably expected to result in a significant injury to the user. A critical component is any component in a life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or system or to affect its safety or effectiveness.

Document Disclaimer

 Information in this publication concerning the devices, applications, or technology described is intended to suggest possible uses and may be superseded. ZILOG, INC. DOES NOT ASSUME LIABILITY FOR OR PROVIDE A REPRESENTATION OF ACCURACY OF THE INFORMATION, DEVICES, OR TECHNOLOGY DESCRIBED IN THIS DOCUMENT. ZILOG ALSO DOES NOT ASSUME LIABILITY FOR INTELLECTUAL PROPERTY INFRINGEMENT RELATED IN ANY MANNER TO USE OF INFORMATION, DEVICES, OR TECHNOLOGY DESCRIBED HEREIN OR OTHERWISE. The information contained within this document has been verified according to the general principles of electrical and mechanical engineering.

Revision History

Each instance in the following revision history table reflects a change to this document from its previous version. For more details, refer to the corresponding pages or appropriate links provided in the table.

Date	Revision Level	Description	Page
Dec 2011	14	Globally updated for the RZK v2.4.0 release.	All
Aug 2010	13	Globally updated for the RZK v2.3.0 release.	All
Sep 2008	12	Globally updated for the RZK v2.2.0 release.	All
Jul 2007	11	Globally updated for proper branding.	All
Jul 2007	10	Globally updated for the RZK v2.1.0 release.	All
Jun 2007	09	Updated document as per Zilog Style Guide; deleted File\Directory\Volume Naming Conventions section.	All
Jul 2006	08	Globally updated for the RZK v2.0.0 release.	All
Apr 2006	07	Globally updated for the RZK v1.2.2 release.	All

Table of Contents

Revision History iii

Introduction v

About This Manual v

Intended Audience v

Manual Organization v

Zilog File System v

File System APIs vi

Related Documents vi

Manual Conventions vii

Safeguards vii

Zilog File System. 1

Zilog File System Architecture 2

Developing Applications with the Zilog File System 4

File System APIs 5

C Run–Time Library Standard Functions 56

Appendix A. Zilog File System Data Types, Macros and Data Structures 77

Zilog File System Data Types 77

Zilog File System Macros 77

Zilog File System Data Structures 79

Appendix B. Zilog File System Error Codes 81

Customer Support 83

Introduction

This Reference Manual describes the APIs associated with the Zilog File System for Zilog's eZ80® CPU-based microprocessors and microcontrollers. This Zilog File System release supports the eZ80Acclaim! family of devices, which includes the eZ80F91 microcontroller.

About This Manual

Zilog recommends that you read and understand the chapters in this manual before using the product. This manual is designed to be used as a reference guide for Zilog File System APIs.

Intended Audience

This document is written for Zilog customers having experience with RTOS and with microprocessors, in writing assembly code, or in writing higher level languages such as C.

Manual Organization

This Reference Manual contains the following chapters and appendices:

Zilog File System

This chapter provides an overview of the Zilog File System architecture and the development of application using Zilog File System.

File System APIs

This chapter describes the Zilog File System APIs.

C Run–Time Library Standard Functions

This chapter describes the C functions supported by the Zilog File System.

Appendix A. Zilog File System Data Types, Macros and Data Structures

This appendix lists Zilog File System data structures.

Appendix B. Zilog File System Error Codes

This appendix lists Zilog File System error codes.

Related Documents

Table 1 lists the related documents that you must be familiar with to use the Zilog File System efficiently.

Table 1. Related Documentation

Document title	Document Number
Zilog File System Quick Start Guide	QS0050
Zilog Real-Time Kernel Quick Start Guide	QS0048
Zilog Real-Time Kernel Reference Manual	RM0006

Manual Conventions

The following conventions are adopted to provide clarity and ease of use.

Courier New Typeface

Code lines and fragments, functions, and various executable files are distinguished from general text by appearing in the Courier New typeface.

For example: `#include "zfsapi.h"`.

Zilog File System Terminology

File System refers to the Zilog File System.

Safeguards

When you use any of Zilog's development platforms, follow the precautions listed below to avoid any damage to the development platform.

Note: Always use a grounding strap to prevent damage resulting from electrostatic discharge (ESD).

Power-Up Precautions

When powering up, observe the following sequence.

1. Apply power to the PC and ensure that it is running properly.
2. Start the terminal emulator program on the PC.
3. Apply power through connector P3 on the eZ80 Development Platform.

Power-Down Precautions

When powering down, observe the following sequence.

1. Exit the monitor program.
2. Remove power from the eZ80 Development Platform.

Zilog File System

The Zilog File System (ZFS) is implemented using the preemptive, multi-tasking Zilog Real-Time Kernel (RZK). ZFS includes drivers compatible with Flash devices from Micron, AMD and Atmel, and allows customers to create new drivers to support other Flash devices.

Key features of the Zilog File System include:

- Implements a core that is independent of the underlying memory device.
- Supports easy configuration of volumes (such as `C:\` or `D:\` drives).
- Provides configuration parameters such as the maximum number of directories to be created and the maximum number of files to be opened at one time. These parameters, related to volume, optimize system operation and serve to consume less memory.
- Supports multiple volume access whether RAM memory, Flash memory, or both memories are employed.
- Implements full-fledged directory operation support.
- Easy system configuration.
- Provides a way to port the Zilog File System core easily to another toolset.
- Supports all basic file and directory operations.
- Supports multiple access to a single file; however, it can be edited by only one person at a time.
- Recovers data after a power failure and implements garbage collection for a Flash device to allow maximum usage of device memory to store files and directories.

- All APIs are multithread safe; i.e, they are reentrant File System APIs.
- Supports the use of a period (.) in filenames or directory names to distinguish between a filename and its extension.
- Supports media error handling; i.e., the recovery of lost data within Flash memory.
- Supports NOR Flash devices.

Zilog File System Architecture

Figure 1 displays the architecture of the Zilog File System.

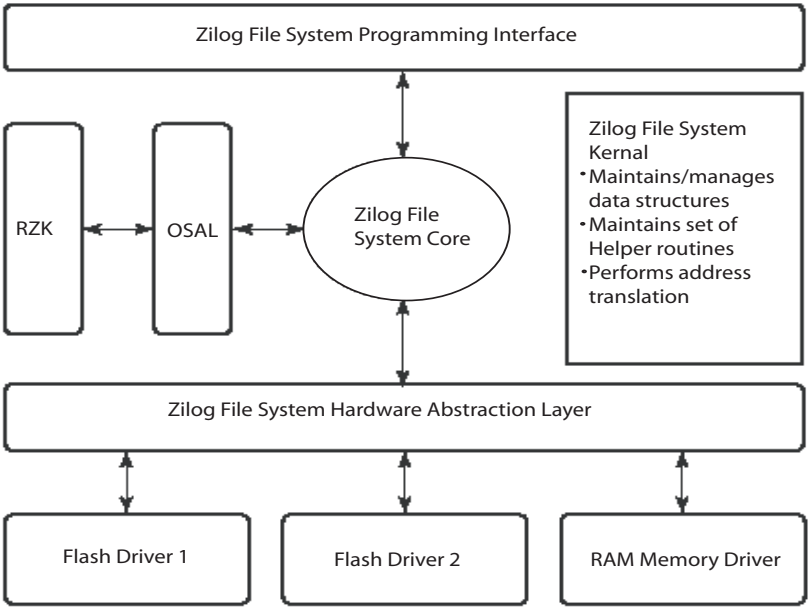

Figure 1. Zilog File System Block Diagram

The architecture of the Zilog File System contains a number of components, each of which is described below.

Zilog File System Programming Interface. This layer contains the API implementation for basic file and directory operations. These interfaces are called by the application to access the files and directories contained in the memory device.

Operating System Abstraction Library (OSAL). This component implements the OS abstraction for the target OS APIs that are used by the Zilog File System.

Zilog File System Core. This layer handles sector-related information such as allocation and deallocation, and performs address translation. It also invokes garbage collection to retrieve dirty sectors from the disk when required.

The Zilog File System implements support for equally-sized blocks, wherein a block refers to a fixed number of bytes present in memory (which is equivalent to a physically-erasable block).

Zilog File System Hardware Abstraction Layer. This layer provides a hardware abstraction (or driver abstraction) layer to integrate multiple devices seamlessly without any changes in other components.

The Zilog File System implements current working directory information on a per-thread basis, not on a per-system basis. However, it implements volumes on a per-system basis. The Zilog File System also allows easy volume configuration.

Developing Applications with the Zilog File System

The application can call any of the Zilog File System APIs. The `ZFSInit()` API must be called first to initialize the system. This API checks the volume for the native format of the Zilog File System and loads file and directory information into memory. The `ZFSInit()` API returns the volume information that is invalid or does not contain a valid

file system. To use these volumes for storing and retrieving files/directories, the caller can check the invalid volumes and format these volumes using the native format of Zilog File System. After the `ZFSInit()` API is executed successfully, the services of Zilog File System become available. You can then create, delete, and rename directories and files.

File System APIs

Table 2 provides a brief description of the Zilog File System Standard APIs. The Zilog File System utilizes these APIs to provide the standard C run–time library file system so that other applications remain unchanged when ported to the ZFS. (These C APIs are listed in Table 3 on page 56.) You can use either these ZFS APIs or the C run-time library standard functions to develop an application, because both perform the same function.

Table 2. Zilog File System Standard APIs

Function Name	Description
ZFSChdir	Changes the current working directory.
ZFSClose	Closes the open file.
ZFSDelete	Deletes an existing file.
ZFSDeleteDir	Deletes an existing directory or subdirectories.
ZFSFormat	Formats the volume for use in Zilog File System.
ZFSGetCwd	Returns the current working directory.
ZFSGetCwdLen	Returns the number of bytes that current working directory string contains.
ZFSGetDirFileCount	Returns the number of files/directories contained in the directory.
ZFSGetErrNum	Returns the error number if recent Zilog File System API execution contains an error.
ZFSGetVolumeCount	Returns the number of volumes contained in the system.
ZFSGetVolumeParams	Returns the system volume parameters.
ZFSInit	Initializes Zilog File System.
ZFSList	Lists all files/directories contained in a path.
ZFSMkdir	Creates a directory under a path.

Table 2. Zilog File System Standard APIs (Continued)

Function Name	Description
ZFSOpen	Opens a file for reading/writing/appending or create a new file.
ZFSRead	Reads data from an open file.
ZFSRename	Renames a file.
ZFSRenameDir	Renames a directory.
ZFSSeek	Sets file read/write pointer to the specified location.
ZFSShutdown	Uninitializes Zilog File System.
ZFSWrite	Writes data to an open file.

ZFSCHDIR

Include

```
#include "zfsapi.h"
```

Prototype

```
ZFS_STATUS_t ZFSChdir(IN INT8 *dir);
```

Description

The `ZFSChdir()` API changes the current working directory to the directory path specified. Further operation is performed on this current working directory if a relative path is provided to any of the Zilog File System APIs. `ZFSInit` must be called before using any of the Zilog File System APIs.

Argument(s)

`dir`	A pointer to a directory name or path to which the current working directory is to be set.

Return Value(s)

This API returns the following values when it is executed.

`ZFSERR_SUCCESS`	Current working directory is changed successfully.
`ZFSERR_NOT_INITIALIZED`	The Zilog File System is not initialized.
`ZFSERR_INVALID_FILEDIR_PATH`	Directory name or directory path is invalid.

ZFSERR_CWD_PATH_LENGTH_MORE	The resultant current working directory string size is more than the configured one.
ZFSERR_INVALID_VOLUME	The volume to which the directory path, filename, and/or current working directory corresponds is invalid. The volume is not formatted with Zilog File System native format or the current working directory information stored for the thread is corrupted.

Example

Change the current working directory to filepath of `C:/root/new_dir`.

```
ZFS_STATUS_t status ;

status = ZFSChdir("C:/root/new_dir");
if(status != ZFSERR_SUCCESS)
 printf("\n Unable to change the directory");
else
 printf("\n Changed the current working directory
successfully");
```

ZFSCLOSE

Include

```
#include "zfsapi.h"
```

Prototype

```
ZFS_STATUS_t ZFSClose(IN ZFS_HANDLE_t handle);
```

Description

The `ZFSClose()` API closes an open file that is associated with the corresponding handle returned by the `ZFSOpen()` API.

Argument(s)

`handle`	The handle of the open file that must be closed.

Return Value(s)

This API returns the following values.

`ZFSERR_SUCCESS`	The file is closed successfully.
`ZFSERR_NOT_INITIALIZED`	The Zilog File System is not initialized. ZFSInit must be called before using any Zilog File System API.
`ZFSERR_INVALID_HANDLE`	The file handle is invalid.

Example

Close the open `new_file.txt` file.

```
ZFS_STATUS_t status ;
extern ZFS_HANDLE_t fs_handle ;
status = ZFSClose(fs_handle);
if(status != ZFSERR_SUCCESS)
```

```
 printf("\n Unable to close the file");
else
 printf("\n Closed the open file");
```

ZFSDELETE

Include

```
#include "zfsapi.h"
```

Prototype

```
ZFS_STATUS_t ZFSDelete(IN INT8 *file_name);
```

Description

The `ZFSDelete()` API deletes the specified file from the disk. The `file_name` argument contains the full path including the filename. The path can be a absolute path or a relative path.

Argument(s)

`file_name`	File path including the filename to be deleted from the disk.

Return Value(s)

This API returns the following values when it is executed.

`ZFSERR_SUCCESS`	Current working directory is changed successfully.
`ZFSERR_NOT_INITIALIZED`	The Zilog File System is not initialized. ZFSInit must be called before using any Zilog File System API.
`ZFSERR_FILE_DIR_DOES_NOT_EXIST`	File does not exist.
`ZFSERR_INVALID_FILEDIR_PATH`	The path is invalid.

ZFSERR_FILE_DIR_IN_USE	File is already open for an operation (reading/writing/ appending).
ZFSERR_INVALID_VOLUME	The volume to which the directory path, filename, and/or current working directory corresponds is invalid. The volume is not formatted with the Zilog File System native format or the current working directory information stored for the thread is corrupted.

Example

Delete the `new_file.txt` file that exists in the `c:/dir1/new_dir` file-path.

```
ZFS_STATUS_t status ;

status = ZFSDelete("C:/dir1/new_dir/new_file.txt");
if(status != ZFSERR_SUCCESS)
 printf("\n Unable to delete the file");
else
 printf("\n Deleted the file");
```

ZFSDELETEDIR

Include

```
#include "zfsapi.h"
```

Prototype

```
ZFS_STATUS_t ZFSDeleteDir(IN INT8 *dir_name, IN UINT8
del_all);
```

Description

The `ZFSDeleteDir()` API deletes a directory and its contents from the disk. The directory name `dir_name` can be provided via an absolute path or relative path. The API returns an error if any threads make the directory or its child directories its current working directory (CWD). It returns an error if any files contained in the directory or its subdirectories are opened for reading or writing. This API returns an error if the root directory is being deleted.

Argument(s)

`dir_name` The directory path or directory name to be deleted from the disk.

`del_all` This argument deletes all the files/directories contained in the directory or deletes only the directory specified from the disk. This argument contains the following values:

- ZFS_FALSE: If `del_all` is set to ZFS_FALSE, it deletes an empty directory. If the directory is not empty, it returns the `ZFSERR_DIRECTORY_NOT_EMPTY` value.
- ZFS_TRUE: If `del_all` is set to ZFS_TRUE, it deletes all files or directories contained in the specified directory including this directory.

Return Value(s)

This API returns the following values when it is executed.

ZFSERR_SUCCESS	The files and directories are deleted when del_all is ZFS_TRUE and the directory is deleted only if it is empty when del_all is ZFS_FALSE.
ZFSERR_NOT_INITIALIZED	ZFS is not initialized. `ZFSInit()` must be called before using any Zilog File System API.
ZFSERR_INVALID_ARGUMENTS	Parameters passed are invalid.
ZFSERR_FILE_DIR_DOES_NOT_EXIST	Directory does not exist.
ZFSERR_INVALID_FILEDIR_PATH	The directory path is invalid.
ZFSERR_INVALID_OPERATION	The directory or its subdirectories are in use (some threads use this directory or its subdirectory as a current working directory).
ZFSERR_DIRECTORY_NOT_EMPTY	The directory is not empty (this error is returned if `del_all` is ZFS_FALSE).
ZFSERR_INVALID_VOLUME	The volume to which the directory path, filename, and/or current working directory corresponds is invalid. The volume is not formatted with the Zilog File System native format or the current working directory information stored for the thread is corrupted.

Example 1

Delete the files and directories contained in the path `C:/dir1/new_dir`.

```
ZFS_STATUS_t status ;
status = ZFSDeleteDir("C:/dir1/new_dir", ZFS_TRUE);
if(status != ZFSERR_SUCCESS)
 printf("\n Unable to delete the files/directories");
else
 printf("\n Deleted the files and directories");
```

Example 2

Delete the `C:/dir1/empty_dir` directory.

```
ZFS_STATUS_t status ;
status = ZFSDeleteDir("C:/dir1/empty_dir", ZFS_FALSE);
if(status != ZFSERR_SUCCESS)
 printf("\n Unable to delete the directory");
else
 printf("\n Deleted the directory");
```

ZFSFORMAT

Include
```
#include "zfsapi.h"
```

Prototype
```
ZFS_STATUS_t ZFSFormat(IN INT8 *volname);
```

Description

The `ZFSFormat()` API formats a volume with the Zilog File System format. If any file contained in the volume is already open, or if any thread has established a current working directory in the volume (except for the root directory of the volume), this API format will not succeed.

Argument(s)

`volname` The name of the volume that must be formatted.

Return Value(s)

This API returns the following values when it is executed.

`ZFSERR_SUCCESS`	The specified volume is formatted successfully.
`ZFSERR_NOT_INITIALIZED`	The Zilog File System is not initialized. `ZFSInit` must be called before using any Zilog File System API.
`ZFSERR_INVALID_VOLUME_NAME`	Volume name is not contained in the configuration.
`ZFSERR_VOLUME_IS_IN_USE`	The volume is in use; that is, a file contained in the volume is open or a thread has established a current working directory within a directory contained in the volume.

Example

Format a volume named C.

```
ZFS_STATUS_t status ;
status = ZFSFormat("C");
if(status != ZFSERR_SUCCESS)
 printf("\n Unable to format the volume");
else
 printf("\n Formatted the volume");
```

ZFSGETCWD

Include

```
#include "zfsapi.h"
```

Prototype

```
ZFS_STATUS_t ZFSGetCwd(IN_OUT INT8 *pcwd_path);
```

Description

The `ZFSGetCwd()` API returns the current working directory as an absolute pathname. The buffer should be allocated by the caller. The caller must call the `ZFSGetCwdLen()` API to obtain the number of bytes of memory to be allocated to store the current working directory.

Argument(s)

`pcwd_path`	A pointer to the memory that is allocated to store the current working directory.

Return Value(s)

This API returns the following values when it is executed.

`ZFSERR_SUCCESS`	The current working directory is returned successfully.
`ZFSERR_NOT_INITIALIZED`	The Zilog File System is not initialized. `ZFSInit` must be called before using any Zilog File System API.
`ZFSERR_INVALID_ARGUMENTS`	Arguments to the API are invalid.

ZFSERR_INVALID_VOLUME	The volume to which the directory path, filename, and/or current working directory corresponds is invalid. The volume is not formatted with the Zilog File System native format or the current working directory information stored for the thread is corrupted.

Example

Retrieve the current working directory.

```
ZFS_STATUS_t status ;
extern INT8 *pcwd_path;             // assuming that
memory is
                                    // allocated
status = ZFSGetCwd(pcwd_path);
if(status != ZFSERR_SUCCESS)
 printf("\n Unable to get the current working
directory path");
else
 printf("\n Current working directory is : %s",
pcwd_path);
```

ZFSGETCWDLEN

Include

```
#include "zfsapi.h"
```

Prototype

```
INT ZFSGetCwdLen(void);
```

Description

The `ZFSGetCwdLen()` API returns the number of bytes of memory to be allocated to store the current working directory. The caller must call `ZFSGetCwd` to obtain the current working directory path.

Argument(s)

None.

Return Value(s)

This API returns the length (positive value) of the current working directory string if successful. If unsuccessful, it returns the following errors:

`ZFSERR_NOT_INITIALIZED`	The Zilog File System is not initialized.
`ZFSERR_INVALID_VOLUME`	The current working directory pointed to the volume is incorrect. The volume is not formatted in the Zilog File System native format or the current working directory information stored for the thread is corrupted.
`ZFSERR_INTERNAL`	Internal error.

Example

Retrieve the current working directory.

```
INT ncwd_len;
ncwd_len = ZFSGetCwdLen();
if( ncwd_len <= 0 )
printf("\n API returned an error");
else
printf("\n Length of the CWD in string format is: %d",
ncwd_len);
```

ZFSGETDIRFILECOUNT

Include

```
#include "zfsapi.h"
```

Prototype

```
INT32 ZFSGetDirFileCount(IN INT8 *dir_path);
```

Description

The `ZFSGetDirFileCount()` API returns the number of files and directories contained in the specified directory. After calling this API, the caller should allocate the memory to obtain details about the files and directories contained in the specified directory by the following command: `(count * sizeof(ZFS_FD_LIST_t))`

Argument(s)

`dir_path`	The directory name or directory path within which the files and directories are counted.

Return Value(s)

This API returns the number of files and directories contained in the directory. If an error occurs, it returns the following values:

`ZFSERR_NOT_INITIALIZED`	The Zilog File System is not initialized. `ZFSInit` must be called before using any Zilog File System API.
`ZFSERR_INVALID_ARGUMENTS`	Arguments to the API are invalid.

ZFSERR_INVALID_FILEDIR_PATH	Directory name or directory path are invalid.
ZFSERR_INVALID_VOLUME	The volume to which the directory path, filename, and/or current working directory corresponds is invalid. The volume is not formatted with the Zilog File System native format or the current working directory information stored for the thread is corrupted.

Example

Obtain the number of files and directories contained in the `C:/dir/new_dir` directory.

```
INT32 nfd_cnt ;

nfd_cnt = ZFSGetDirFileCount("C:/dir/new_dir");
if(nfd_cnt >= 0)
 printf("\n Number of files/dirs = %ld", nfd_cnt);
else
 printf("\n Error occurred in the execution");
```

ZFSGETERRNUM

Include

```
#include "zfsapi.h"
```

Prototype

```
ZFS_STATUS_t ZFSGetErrNum(void);
```

Description

The `ZFSGetErrNum()` API returns the error code stored when a ZFS API is executed. The return values are valid only if this API is called immediately after either the `ZFSOpen()` or `fopen()` APIs are called.

Argument(s)

None.

Return Value(s)

This API returns the status of the previously-executed API and is valid only for `ZFSOpen` or `fopen` APIs. See <CrossRef>ZFSOpen on page 37.

Example

Obtain the error number for the previous operation in the current thread.

```
ZFS_STATUS_t status ;
ZFS_HANDLE_t fs_handle ;
fs_handle = ZFSOpen("C:/dir/child_dir/bin_file.txt",
ZFS_WRITE, ZFS_MODE_BINARY);
if(fs_handle == NULL)
{
 printf("\n File open error and error number is : %d",
ZFSGetErrNum());
}
else
 printf("\n File is opened in WRITE mode");
```

ZFSGETVOLUMECOUNT

Include

```
#include "zfsapi.h"
```

Prototype

```
ZFS_STATUS_t ZFSGetVolumeCount(void);
```

Description

The `ZFSGetVolumeCount()` API returns the number of volumes contained in the system.

Argument(s)

None.

Return Value(s)

This API returns number of volumes contained in the system.

Example

Obtain the number of volumes contained in the system.

```
ZFS_STATUS_t status ;

status = ZFSGetVolumeCount();
if(status < 0)
 printf("\n Error");
else
 printf("\n Volume count = %d", status);
```

ZFSGETVOLUMEPARAMS

Include

```
#include "zfsapi.h"
```

Prototype

```
ZFS_STATUS_t ZFSGetVolumeParams(IN INT8 *vol_name,
ZFS_VOL_PARAMS_t *vol_params, UINT8 get_all);
```

Description

The `ZFSGetVolumeParams()` API returns the volume parameters volume name, total space, free space, dirty space, and used space. An option provides the details about all volumes contained in the system, or about a particular volume. The caller of this API should first call the `ZFSGetVolumeCount` API to obtain the number of volumes present and then allocate memory (`count * sizeof(ZFS_VOL_PARAMS_t)`). The starting location of the allocated memory should be passed to this API.

Argument(s)

`vol_name`	Volume name for which volume information is obtained. This name can be anything except `NULL` if `get_all` is equal to ZFS_TRUE; in this case, all volume information is returned.

`vol_params`	Pointer to the first location of the memory that was allocated to store the volume information.
`get_all`	This flag specifies whether to obtain information for all volumes or information specific to one volume. It can include either of the values specified below: • ZFS_TRUE: If this flag is specified, then information about all volumes is returned. In this case, the vol_name can be anything except `NULL`. • ZFS_FALSE: If this flag is specified, then information about only the specified volume is returned.

Return Value(s)

This API returns the following values when it is executed.

`ZFSERR_SUCCESS`	Volume information is returned successfully.
`ZFSERR_NOT_INITIALIZED`	The Zilog File System is not initialized. `ZFSInit` must be called before using any Zilog File System API.
`ZFSERR_INVALID_ARGUMENTS`	Arguments to the API are invalid.

Example

Display information for volumes contained in the system.

```
ZFS_STATUS_t status;
extern ZFS_HANDLE_t fs_handle ;
extern ZFS_VOL_PARAMS_t vol_params[];// assuming that
                                    // memory is
                                    // allocated.
status = ZFSGetVolumeParams("EXTF", &vol_params[0],
ZFS_TRUE);
```

```
if(status != ZFSERR_SUCCESS)
 printf("\n Error returned");
else
{
// display volume information here
}
```

ZFSINIT

Include

```
#include "zfsapi.h"
```

Prototype

```
ZFS_STATUS_t ZFSInit( PZFS_VOL_PARAMS_t pvol_params );
```

Description

This API initializes Zilog File System, and retrieves information stored in the volume. If the `ZFSInit` API is successful in retrieving information, it updates the internal structure of the Zilog File System with appropriate information so that all other ZFS APIs work. If the configured volume is invalid, the `ZFSInit` API returns the number of invalid volumes located in the configuration, the details of which are stored in the `pvol_params` argument. The caller must first call the `ZFSGetVolumeCount()` API, then allocate memory for the ZFS_VOL_PARAMS_t structure in all volumes, and pass the address to the `ZFSInit` API. If a volume is invalid, the volume can be made valid only by calling the `ZFSFormat()` API and formatting the volume using the Zilog File System native format. The value of the `ZFS_VOL_PARAMS_t.is_valid` API indicates the validity of the volume.

Argument(s)

`pvol_params`	Address of the memory where the details for invalid volumes are stored.

Return Value(s)

This API returns the number of invalid volumes present in the configuration if successful. Otherwise, it returns the one of the following values:

ZFSERR_SUCCESS	Initialization of the Zilog File System is successful.
Number of invalid volumes	The ZFSInit API returns the number of invalid volumes located in the configuration, the details of which are stored in the pvol_params argument.
ZFSERR_ALREADY_INITIALIZED	The Zilog File System is already initialized.

Example

Initialize the File System.

```
extern ZFS_VOL_PARAMS_t vol_params[] ; // Assuming the
                                       // memory is
                                       // allocated.
ZFS_STATUS_t nInvalidVolCnt ;
ZFS_STATUS_t nVols ;

nVols = ZFSGetVolumeCount() ; // The memory is
                    //allocated ZFS_VOL_PARAMS_t
                    //structure for number of volumes
nInvalidVolCnt = ZFSInit ( &vol_params[0] ) ;
if( nInvalidVolCnt == 0 )
printf("\nZilog File System is initialized
successful");
else if( nInvalidVolCnt < 0 )
printf("\nError in executing the API");
else
printf("\nThe system has invalid volumes : %d",
nInvalidVolCnt) ;
```

ZFSLIST

Include

```
#include "zfsapi.h"
```

Prototype

```
ZFS_STATUS_t ZFSList(IN INT8 * path, IN_OUT
ZFS_FD_LIST_t * list , IN UINT8 startCnt);
```

Description

The `ZFSList()` API returns file and directory information, and returns a maximum of eight entries each time it is called. If the directory contains more than eight files or directories, this API must be called until an error is received or less than eight items are received. The caller must provide an entry count from which eight items will be returned.

The caller of this API must call first the `ZFSGetDirFileCount()` API to obtain the number of files/directories present and then allocate the memory `(count * sizeof(ZFS_FD_LIST_t))`. The starting location of the allocated memory should be passed to this API.

Argument(s)

`path`	Directory name or directory path for which file and directory information is to be retrieved.
`list`	Pointer to the first location of the memory that is allocated to store file or directory data.
`startCnt`	Entry count after which eight new items are returned.

Return Value(s)

This API returns one of the following values when it is executed.

`ZFSERR_SUCCESS`	File and directory data is successfully retrieved and stored in the argument.
`ZFSERR_NOT_INITIALIZED`	The Zilog File System is not initialized. `ZFSInit` must be called before using any Zilog File System API.
`ZFSERR_INVALID_ARGUMENTS`	Arguments to the API are invalid
`ZFSERR_INVALID_FILEDIR_PATH`	Directory path or directory name is invalid.
`ZFSERR_FILE_DIR_DOES_NOT_EXIST`	Directory name does not exist in the volume.
`ZFSERR_INVALID_OFFSET_RANGE`	Count of entry is invalid.
`ZFSERR_INVALID_VOLUME`	The volume to which the directory path, filename, and/or current working directory corresponds is invalid. The volume is not formatted with the Zilog File System native format or the current working directory information stored for the thread is corrupted.

Example

Display file and directory information about the files and directories contained in the `C:/dir/child_dir`.

```
ZFS_STATUS_t status ;
extern ZFS_FD_LIST_t fd_list[];     // assuming that
                                    // memory is
                                    // allocated.
ZFS_STATUS_t cnt = 0 ;
do
{
status = ZFSList("C:/dir/child_dir", &fd_list[0],
cnt);
if(status != ZFSERR_SUCCESS)
 printf("\n Error returned");
else
{
 //lists directories/files
}
cnt = cnt+status ;
} while(status > 0);
```

ZFSMKDIR

Include

```
#include "zfsapi.h"
```

Prototype

```
ZFS_STATUS_t ZFSMkdir(IN INT8 *path, IN INT8
*dirname);
```

Description

The `ZFSMkdir()` API creates a directory in the specified directory path. The path can be a relative path or an absolute path wherein the new directory name, `dirname`, must be only a directory name and must not contain a pathname.

Argument(s)

`path`	Path within which the new directory must be created.
`dirname`	Directory name to be created.

Return Value(s)

This API returns one of the following values when it is executed.

`ZFSERR_SUCCESS`	New directory is created successfully.
`ZFSERR_NOT_INITIALIZED`	The Zilog File System is not initialized. `ZFSInit` must be called before using any Zilog File System API.
`ZFSERR_INVALID_ARGUMENTS`	Arguments to the API are invalid.

ZFSERR_INVALID_FILEDIR_PATH	Directory path or directory name is invalid.
ZFSERR_INVALID_FILE_DIR_NAME	Directory that must be created is invalid.
ZFSERR_FILE_DIR_DOES_NOT_EXIST	Directory name within which the new directory must be created does not exist in the volume.
ZFSERR_FILE_DIR_ALREADY_EXISTS	A file or directory with the same name of new directory to be created already exists in the volume.
ZFSERR_FILE_DIR_COUNT_LIMIT_REACHED	The directory in which a new directory to be created already contains the maximum number of files and/or directories (255 files).
ZFSERR_DIR_COUNT_LIMIT_REACHED	Directory count limit throughout the system is reached.
ZFSERR_INTERNAL	Internal error.

ZFSERR_DATAMEDIA_FULL	Volume is full. No space is present to create a directory.
ZFSERR_INVALID_VOLUME	The volume to which the directory path, filename, and/or current working directory corresponds is invalid. The volume is not formatted with the Zilog File System native format or the current working directory information stored for the thread is corrupted.

Example

Create a directory named `new_child_dir` within the `C:/dir/child_dir` directory.

```
ZFS_STATUS_t status ;
status = ZFSMkdir("C:/dir/child_dir",
"new_child_dir");
if(status != ZFSERR_SUCCESS)
 printf("\n New directory is created successfully");
else
 printf("\n Unable to create a new directory");
```

ZFSOPEN

Include

```
#include "zfsapi.h"
```

Prototype

```
ZFS_HANDLE_t ZFSOpen(IN INT8 *filename, IN UINT8 mode,
IN UINT8 type);
```

Description

The `ZFSOpen()` API opens the existing file in read, or read/write, or append mode. The `ZFSOpen()` API creates a new file if the specified file does not exist. The corresponding `ZFSRead()` or `ZFSWrite()` API can be used to read from or write to the file for data. On successful execution, this API returns the handle for the file that must be used when reading from or writing to the file. The filename can be in a relative path or an absolute path, and can include the name of the file that must be opened. Reading and writing of the file can be performed in ASCII or BINARY mode. A file can be opened in `READ` mode many times, but a file in the `WRITE/READ_WRITE/APPEND` mode can be opened in only one instance.

Argument(s)

`filename`	File name that must be opened in respective mode.
`mode`	File opening mode can contain any of the following values: ZFS_READ: Opens file in read mode; to be successful, the file should exist in the volume. The file position is set to the beginning. ZFS_WRITE: Opens file in write mode; If the file does not exist in the volume, then this API creates a new file; otherwise, if the file exists, the API truncates the file size to 0. The file position is set to the beginning. ZFS_READ_WRITE: The file is opened in read/write mode. To be successful, the file should exist in the volume. The file position is set to the beginning of the file. ZFS_APPEND: The file is opened in append mode. Only a write operation can be performed on this file handle. Data is appended to the end of the file.
`type`	File can be opened in translation or no-translation mode and can contain either of the following values: ZFS_MODE_BINARY: File opened in binary mode; no translation occurs while reading or writing. ZFS_MODE_ASCII: Translation occurs when file is opened in ASCII mode. While reading, the carriage return-line feed (CR-LF) combinations are translated to line feed (LF) and can be read until the number of bytes in the `ZFSRead` API are reached. While writing, the LF character is converted into the (CR-LF) combination.

Return Value(s)

This API returns the handle for the file that is opened. This handle is provided to the `ZFSRead()`, `ZFSWrite()`, `ZFSSeek()`, and `ZFSClose()`

APIs. If file opening is not successful, the API returns NULL. The error code can be obtained by calling the `ZFSGetErrNum` API, which returns one of the following values:

`ZFSERR_SUCCESS`	New directory is created successfully.
`ZFSERR_NOT_INITIALIZED`	The Zilog File System is not initialized. `ZFSInit` must be called before using any of the Zilog File System APIs.
`ZFSERR_INVALID_ARGUMENTS`	Arguments to the API are invalid.
`ZFSERR_INVALID_FILEDIR_PATH`	File path or file name are invalid.
`ZFSERR_INVALID_FILE_DIR_NAME`	Invalid file name or directory name.
`ZFSERR_FILE_DIR_DOES_NOT_EXIST`	The directory in which the file resides, or the file has to be created, do not exist.
`ZFSERR_FILE_IS_ALREADY_OPEN`	The file trying to open is already opened by the same thread or other threads in `READ/WRITE/ READ_WRITE/APPEND` mode.
`ZFSERR_MAX_FILE_OPEN_COUNT_ REACHED`	File open instance count limit is reached. There are already a number of file open instances that are equal to the number that is configured.

ZFSERR_FILE_DIR_COUNT_LIMIT_REACHED	Number of files and directories contained in the directory has already reached the maximum limit of 255 files.
ZFSERR_DATAMEDIA_FULL	Volume is full. No space is present to create a file.
ZFSERR_INVALID_VOLUME	The volume to which the directory path, filename, and/or current working directory corresponds is invalid. The volume is not formatted with the Zilog File System native format or the current working directory information stored for the thread is corrupted.

Example

Create a file named `bin_file.txt` in the `C:/dir/child_dir` directory.

```
ZFS_STATUS_t status ;
ZFS_HANDLE_t fs_handle ;
fs_handle = ZFSOpen("C:/dir/child_dir/bin_file.txt",
ZFS_WRITE, ZFS_MODE_BINARY);
if(fs_handle == NULL)
{
 printf("\n File open error and error number is : %d",
ZFSGetErrNum());
}
else
 printf("\n File is opened in WRITE mode");
```

ZFSREAD

Include

```
#include "zfsapi.h"
```

Prototype

```
INT32 ZFSRead(IN ZFS_HANDLE_t handle, IN_OUT UINT8
*buf, IN UINT bytes);
```

Description

The `ZFSRead()` API reads characters up to a specific number of bytes from the file that is associated with the handle. The file pointer associated with the handle is increased by the number of bytes actually read. If the stream is opened in ASCII mode, carriage return-line feed (CR-LF) pairs are replaced with single line feed (LF) characters. The replacement has no effect on the file pointer or return value. The position of the file pointer is indeterminate if an error occurs.

Argument(s)

`handle`	Handle of the file upon which a read operation is to be performed.
`buf`	Buffer to store the read data. The caller of this API should allocate memory for sufficient bytes.
`bytes`	Number of characters to read from the file.

Return Value(s)

This API returns the number of bytes read if successful; otherwise, it returns one of the following error values (these error codes are negative):

`ZFSERR_NOT_INITIALIZED`	The Zilog File System is not initialized. `ZFSInit` must be called before using any Zilog File System API.
`ZFSERR_INVALID_ARGUMENTS`	Arguments to the API are invalid.
`ZFSERR_INVALID_HANDLE`	Handle is not associated with the opening of the file.
`ZFSERR_INVALID_OPERATION`	The Read operation is invalid on the opening of the file (the file is not opened in ZFS_READ or ZFS_READ_WRITE mode) or the file position is already reached at the end of the file.

Example

Use the `ZFSRead()` API to read 100 bytes from a file that is opened in ZFS_READ mode then contain the corresponding handle in `fs_handle`.

```
ZFS_HANDLE_t fs_handle;
ZFS_STATUS_t status ;
INT32 numBytesRead ;
UINT8 buf_read[ 100 ] ;
numBytesRead = ZFSRead(fs_handle, &buf_read[0], 100);
if(numBytesRead <= 0)
 printf("\n Read error");
else
 printf("\n Read %ld bytes from the file",
numBytesRead);
```

ZFSRENAME

Include

```
#include "zfsapi.h"
```

Prototype

```
ZFS_STATUS_t ZFSRename(IN INT8 *src_file_path, IN INT8
*dst_file_name);
```

Description

The `ZFSRename()` API renames a file with a new file name. The file-name that must be renamed can be in a relative path or an absolute path. If the file to be renamed is in use (that is, if the file is opened by any thread for reading/writing), then this API returns an error.

Argument(s)

`src_file_path`	The name of the file that must be renamed. The specified path can include the file name and can be a relative or absolute path.
`dst_file_name`	New name for the file.

Return Value(s)

This API returns one of the following values when it is executed.

`ZFSERR_SUCCESS`	File is renamed to the new name successfully.
`ZFSERR_NOT_INITIALIZED`	The Zilog File System is not initialized. ZFSInit must be called before using any Zilog File System API.

ZFSERR_INVALID_ARGUMENTS	Arguments to the API are invalid.
ZFSERR_INVALID_FILEDIR_PATH	File path is invalid.
ZFSERR_INVALID_FILE_DIR_NAME	New file name is invalid.
ZFSERR_FILE_DIR_DOES_NOT_EXIST	The file that must be renamed does not exist in the directory path.
ZFSERR_FILE_DIR_ALREADY_EXISTS	A file or directory with the same name as the new file name already exists in the volume.
ZFSERR_FILE_DIR_IN_USE	The file that must be used is already open for reading/ writing/appending.
ZFSERR_INTERNAL	Internal error.
ZFSERR_DATAMEDIA_FULL	Volume is full. No space is available to create a directory.
ZFSERR_INVALID_VOLUME	The volume to which the directory path, filename, and/or current working directory corresponds is invalid. The volume is not formatted with the Zilog File System native format or the current working directory information stored for the thread is corrupted.

Example

Use the `ZFSRename()` API to rename a file named `old_file_name` with a name `new_file_name` in the `C:/dir/child_dir` directory.

```
ZFS_STATUS_t status ;
status = ZFSRename("C:/dir/child_dir/old_file_name",
"new_file_name");
if(status != ZFSERR_SUCCESS)
 printf("\n File is renamed to the new file name");
else
 printf("\n Unable to rename file name");
```

ZFSRENAMEDIR

Include

```
#include "zfsapi.h"
```

Prototype

```
ZFS_STATUS_t ZFSRenameDir(IN INT8 *src_dir_path, IN
INT8* dst_dir_name);
```

Description

The `ZFSRenameDir()` API renames a directory. The new directory name can be in a relative path or an absolute path. If the directory to be renamed is in use (that is, if a file is opened for operation such as reading/writing in this directory or subdirectory, or if any thread adopts this directory or subdirectory as the current working directory), then this API returns an error. This API does not allow the root directory to be renamed.

Argument(s)

`src_dir_path`	The directory name that must be renamed. The path includes the directory name and can be relative or absolute path.
`dst_dir_name`	New directory name for the directory.

Return Value(s)

This API returns one of the following values when it is executed.

`ZFSERR_SUCCESS`	Directory is renamed successfully.
`ZFSERR_NOT_INITIALIZED`	The Zilog File System is not initialized. ZFSInit() must be called before using any Zilog File System API.

ZFSERR_INVALID_ARGUMENTS	Arguments to the API are invalid.
ZFSERR_INVALID_FILEDIR_PATH	Directory path is invalid.
ZFSERR_INVALID_FILE_DIR_NAME	New directory name is invalid.
ZFSERR_FILE_DIR_DOES_NOT_EXIST	The directory that must be renamed does not exist in the volume.
ZFSERR_FILE_DIR_ALREADY_EXISTS	A file or directory with the same name as that of the new directory already exists in the volume.
ZFSERR_FILE_DIR_IN_USE	Some files in the directory are already open for use, or the child directories of the specified directory that must be renamed are set as the current working directory of a thread.
ZFSERR_INTERNAL	Internal error.
ZFSERR_INVALID_VOLUME	The volume to which the directory path, filename, and/or current working directory corresponds is invalid. The volume is not formatted with the Zilog File System native format or the current working directory information stored for the thread is corrupted.

Example

Use the `ZFSRenameDir` API to rename a directory named `old_dir_name` with the new directory name `new_dir_name` in the `C:/dir/child_dir` directory.

```
ZFS_STATUS_t status ;
status = ZFSRenameDir("C:/dir/child_dir/old_dir_name",
"new_dir_name");
if(status != ZFSERR_SUCCESS)
 printf("\n Directory is renamed to the new directory
name");
else
 printf("\n Unable to rename directory");
```

ZFSSEEK

Include

```
#include "zfsapi.h"
```

Prototype

```
ZFS_STATUS_t ZFSSeek(IN ZFS_HANDLE_t handle, IN INT32
offset, IN INT8 origin);
```

Description

The `ZFSSeek()` API moves the file pointer to the specified value. The next operation on the file occurs at a new location. This API can be used to reposition the pointer anywhere in the file.

Note: If this API is called on a file that is opened in ZFS_APPEND mode, it returns an error.

Argument(s)

`handle`	Handle of the file on which a seek operation is to be performed.
`offset`	Number of bytes from the origin.
`origin`	Initial position from where the offset must be moved. This parameter can contain any of the following values: ZFS_FILE_BEGIN: origin is from the beginning of the file. ZFS_FILE_END: origin is from the end of the file. ZFS_FILE_CURRENT: origin is from the current file pointer position.

Return Value(s)

This API returns one of the following values when it is executed.

`ZFSERR_SUCCESS`	Seek operation is performed successfully.
`ZFSERR_NOT_INITIALIZED`	The Zilog File System is not initialized. ZFSInit must be called before using any Zilog File System API.
`ZFSERR_INVALID_ARGUMENTS`	Arguments to the API are invalid.
`ZFSERR_INVALID_HANDLE`	Handle is not associated with the opening of the file.
`ZFSERR_INVALID_eOPERATION`	The seek operation is invalid because it is being performed on a file that is open in ZFS_APPEND mode.
`ZFSERR_INVALID_OFFSET_RANGE`	The file pointer position movement exceeds the file size.

Example

Use the `ZFSSeek()` API to move the file pointer position to 100 from the current position for the file `newfile.txt`. This file is open in READ mode, and its handle is contained in `fs_handle`.

```
ZFS_STATUS_t status ;
ZFS_HANDLE_t fs_handle ;
status = ZFSSeek(fs_handle, 100, ZFS_FILE_CURRENT);
if(status != ZFSERR_SUCCESS)
 printf("\n Seek operation returned an error");
else
```

```
 printf("\n Seek operation is performed
successfully");
```

ZFSSHUTDOWN

Include

```
#include "zfsapi.h"
```

Prototype

```
ZFS_STATUS_t ZFSShutdown(void);
```

Description

The `ZFSShutdown()` API uninitializes the File System for all volumes contained in the configuration. The `ZFSInit` API must be called to reinitialize the File System.

Argument(s)

None.

Return Value(s)

This API returns one of the following values:

`ZFSERR_SUCCESS`	The Zilog File System has shut down successfully.
`ZFSERR_ALREADY_SHUTDOWN`	The Zilog File System has already shut down.
`ZFSERR_FS_BUSY`	If any thread opens a file for reading/ writing, this API returns an error.

Example

Shut down the ZFS.

```
ZFS_STATUS_t status ;
status = ZFSShutdown();
if(status != ZFSERR_SUCCESS)
 printf("\n File system shut down is not successful");
```

```
else
 printf("\n File system shutdown is successful");
```

ZFSWRITE

Include

```
#include "zfsapi.h"
```

Prototype

```
INT32 ZFSWrite(IN ZFS_HANDLE_t handle, IN UINT8 *buf,
IN UINT bytes);
```

Description

The `ZFSWrite()` API writes characters up to a specific number of bytes into a file. The file pointer associated with the handle is incremented by the number of bytes actually written. If the file is opened in ASCII mode, each carriage return (CR) is replaced with a carriage return-line feed (CR-LF) combination. The replacement has no effect on the return value.

Argument(s)

`handle`	Handle of the file on which the write operation is to be performed.
`buf`	Data to write into the file.
`bytes`	Number of characters to write into the file.

Return Value(s)

This API returns the number of bytes written if successful; otherwise, it returns one of the following values to indicate an error (these error codes are negative):

`ZFSERR_NOT_INITIALIZED`	The Zilog File System is not initialized. ZFSInit must be called before using any Zilog File System API.
`ZFSERR_INVALID_ARGUMENTS`	Arguments to the API are invalid.

ZFSERR_INVALID_HANDLE	Handle is not associated with the file open instance.
ZFSERR_INVALID_OPERATION	The Write operation is invalid on the instance of file open (indicates that the file is not opened in ZFS_WRITE or ZFS_READ_WRITE or ZFS_APPEND mode).
ZFSERR_DATAMEDIA_FULL	No empty space is available in the volume to perform the write operation.
ZFSERR_DEVICE	Device returned an error.

Example

Write 100 bytes to a file that is opened in ZFS_READ_WRITE mode. The file's handle is contained in `fs_handle`.

```
ZFS_STATUS_t status ;
extern ZFS_HANDLE_t fs_handle ;
INT32 numBytesWritten ;
UINT8 buf_write[ 100 ] ;// this buffer contains data
                        // to write
numBytesWritten =
ZFSWrite(fs_handle,&buf_write[0],100);
if(numBytesWritten <= 0)
 printf("\n Read error");
else
 printf("\n Written %ld bytes to the file",
numBytesWritten);
```

C Run–Time Library Standard Functions

The standard library of C functions supported by the Zilog File System is listed in Table 3.

Table 3. Zilog File System: Supported C Standard Library APIs

Function Name	Description
fopen	Opens a file for reading/writing.
fclose	Closes an open file.
fread	Reads the specified number of bytes from the file.
fwrite	Writes the specified number of bytes into the file.
fgetc	Returns a character from the file.
fputc	Returns a character into the file.
fgets	Returns a string from the file.
fputs	Stores a string into the file.
fseek	Alters the file pointer position.
ftell	Returns the file pointer position.
feof	Determines whether it is end of file or not.

FOPEN

Include

```
#include "cfileapi.h"
```

Prototype

```
FILE *fopen(const char *filename, const char *mode);
```

Description

The `fopen()` function opens a file specified by a filename with the type of access defined by mode. If successful, this function returns a handle for the file; otherwise, it returns `NULL`. If the file is opened in ASCII or translated mode, then, when reading from the file, each carriage return-line feed (CR-LF) character pair will be translated into a line feed (LF) character. During writing, each LF character is converted into a CR-LF pair.

Argument(s)

`filename`	The name of the file that must be opened. This parameter can also contain the path in which the file is contained. The path can be relative to the current working directory or an absolute path.
`mode`	File opening mode can contain any of the following values: **r** Opens specified file in read mode. If the file does not exist or cannot be found, the fopen call fails (translation of new line characters). **w** Opens an empty file in write mode. If the file exists, the file size is truncated to zero (translation of new line characters).

`mode (cont'd)`

a Opens specified file in append mode, before writing the new data to the file; if the specified file does not exist, a new file is created with the specified file name and opened in append mode (translation of new line characters).

r+ Opens in both read and write mode (the file must exist; translation of new line characters).

rb Opens for reading. If the file does not exist or cannot be found, the fopen call fails (translation of new line characters is suppressed).

wb Opens an empty file for writing. If the file exists, the file size is truncated to zero (translation of new line characters is suppressed).

ab Opens for writing at the end of the file (appending) before writing new data to the file; if the specified file does not exist, a new file is created with the specified file name and opened in append mode (translation of new line characters is suppressed).

r+b Opens for both reading and writing (the file must exist; translation of new line characters is suppressed).

Return Value(s)

This function returns a handle to the file that is opened if successful; otherwise it returns `NULL`.

Example

Use the `fopen()` function to open a file called `new_file.txt` in which the file handle is stored in `fs_handle`.

```
struct FILE *fs_handle ;
```

```
fs_handle = fopen("new_file.txt", "wb");
if(fs_handle == NULL)
 printf("\n File cannot be opened for writing");
else
 printf("\n File opened in WRITE mode successfully");
```

FCLOSE

Include
```
#include "cfileapi.h"
```

Prototype
```
int fclose(FILE *stream);
```

Description
The `fclose()` function closes an open file.

Argument(s)
`stream` Handle for the file that must be closed.

Return Value(s)
This function returns 0 if the file is closed successfully; otherwise, it returns -1 to indicate an error.

Example
Use the `fclose()` function to close the file called `new_file.txt` in which in the handle is stored in `fs_handle`.

```
extern struct FILE *fs_handle;// file handle for
                              // "new_file.txt"

if(fclose(fs_handle) != 0)
 printf("\n File could not be closed.");
else
 printf("\n File closed successfully");
```

FREAD

Include

```
#include "cfileapi.h"
```

Prototype

```
size_t fread(void *buffer, size_t size, size_t count,
FILE *stream);
```

Description

The `fread()` function reads data up to the `count` items of the specified `size` bytes from the input stream and stores them in a buffer. The file pointer associated with the stream, if any, is increased by the number of bytes actually read. If the stream is opened in text mode (ASCII or translated), carriage return-line feed (CR-LF) pairs are replaced with single line feed characters. The replacement has no effect on the file pointer or the return value. The file-pointer position is indeterminate if an error occurs.

Argument(s)

`buffer`	Storage location for the data.
`size`	Item size in bytes.
`count`	Maximum number of items to be read.
`stream`	Pointer to the FILE structure (file upon which the read operation is to be performed).

Return Value(s)

The `fread()` function returns the number of items actually read, which may be less than `count` if an error occurs or if the end of the file is encountered before reaching `count`. Use the `feof` function to distinguish

a read error from an end-of-file condition. If an error occurs, this function returns 0.

Example

Use the `fread()` function to read 1000 items of 16 bytes' length from the file named `new_file.txt` in which the handle is stored in `fs_handle`.

```
extern struct FILE *fs_handle;      // file handle for
                                    //"new_file.txt"
unsigned char buf[ 1000 * 16 ] ;
size_t cnt ;
cnt = fread(&buf[0], 16, 1000, fs_handle);
if(cnt == 0)
 printf("\n unable to read the contents or an error
has occurred");
else
 printf("\n Items read = %d", cnt);
```

FWRITE

Include

```
#include "cfileapi.h"
```

Prototype

```
size_t fwrite(const void *buffer, size_t size, size_t
count, FILE *stream);
```

Description

The `fwrite()` function writes data up to `count` items, of specified `size` length each, from the buffer to the output stream. The file pointer associated with stream is incremented by the number of bytes actually written. If stream is opened in text mode (ASCII or translated), each carriage return is replaced with a carriage return/line feed pair. The replacement has no effect on the return value.

Argument(s)

`buffer`	Pointer to data to be written
`size`	Item size in bytes
`count`	Maximum number of items to be written
`stream`	Pointer to the FILE structure (the file on which the Write operation is to be performed)

Return Value(s)

The `fwrite()` function returns the number of items actually written, which may be less than `count` if an error occurs. In addition, if an error occurs, the file position indicator cannot be determined.

Example

Use the `fwrite` function to write 1000 items of 16 bytes' length to a file named `new_file.txt` for which the handle is stored in `fs_handle`.

```
extern struct FILE *fs_handle ; // file handle for
                                      //"new_file.txt"
extern unsigned char buf[ 1000 * 16 ] ; // buffer to
                                       //write
size_t cnt ;
cnt = fwrite(&buf[0], 16, 1000, fs_handle);
if(cnt == 0)
 printf("\n unable to write the contents or an error
has occurred");
else
 printf("\n Items written to the file = %d", cnt);
```

FGETC

Include

```
#include "cfileapi.h"
```

Prototype

```
int fgetc(FILE *stream);
```

Description

This function reads a single character from the current position of a file associated with `stream`. The function increments the associated file pointer to point to the next character. If the stream is at end of file, the end-of-file indicator for the stream is set.

Argument(s)

`stream`	Pointer to the FILE structure (file on which the Read operation is to be performed).

Return Value(s)

The `fgetc()` function returns the character read as an `int` if sucessfully executed and returns `EOF` to indicate an error or the end of the file.

Example

Use the `fgetc()` function to read a character from a file named `new_file.txt` in which the handle is stored in `fs_handle`.

```
extern struct FILE *fs_handle ; // file handle for
                                //"new_file.txt"
int char_read ;
char_read = fgetc(fs_handle);
if(char_read == EOF)
 printf("\n unable to read the character");
else
```

```
printf("\n Character read was : %c", char_read);
```

FPUTC

Include

```
#include "cfileapi.h"
```

Prototype

```
int fputc(int c, FILE *stream);
```

Description

The `fputc()` function writes the single character `c` to a file at the position indicated by the associated file position indicator (if defined) and advances the indicator as appropriate. If the file was opened in append mode, the character is appended to the end of the stream.

Argument(s)

`c`	Character to write into the file.
`stream`	Pointer to the FILE structure (file on which the Write operation is to be performed).

Return Value(s)

This function returns the character written. In the event of an error, EOF is returned.

Example

Use the `fputc` function to write a character to a file named `new_file.txt` in which the handle is stored in `fs_handle`.

```
extern struct FILE *fs_handle ; // file handle for
                                // "new_file.txt"
int char_written ;

char_written = fputc('A', fs_handle);
```

```
if(char_written == EOF)
 printf("\n unable to write the character");
else
 printf("\n Character written was : %c",
char_written);
```

FGETS

Include

```
#include "cfileapi.h"
```

Prototype

```
char *fgets(char *string, int n, FILE *stream);
```

Description

The `fgets()` function reads a string from the input stream argument and stores it in `string`. `fgets` reads characters from the current stream position. It includes the first newline character at the end of the stream, or includes the number of characters read up to n–1, whichever occurs first. The result stored in `string` is appended with a NULL character. The newline character, if read, is included in the string.

Argument(s)

`string`	Storage location for the read data.
`n`	Maximum number of characters to read.
`stream`	Pointer to the FILE structure (file on which the Read operation is to be performed).

Return Value(s)

Each of these functions returns `string`. `NULL` is returned to indicate an error or an end-of-file condition. Use the `feof` function to determine whether an error occurred.

Example

Use `fgets()` to read a string from a file named `new_file.txt` in which the handle is stored in `fs_handle`.

```
extern struct FILE *fs_handle ; // file handle for
                                        // "new_file.txt"
extern char *pbuf;
pbuf = fgets(pbuf, 100, fs_handle);
if(pbuf == NULL)
 printf("\n unable to read the string");
else
 printf("\n String read is : %s", pbuf);
```

FPUTS

Include

```
#include "cfileapi.h"
```

Prototype

```
int fputs(const char *string, FILE *stream);
```

Description

The `fputs()` function copies `string` to the output stream at the current position and does not copy the terminating `NULL` character.

Argument(s)

`string`	String that must be written to the file.
`stream`	Pointer to the FILE structure (file on which the write operation is to be performed).

Return Value(s)

This function returns a non-negative value if it is successful (excluding 0); otherwise, it returns EOF.

Example

Use the `fputs()` function to write the string `Hello World` to a file named `new_file.txt` in which the handle is stored in `fs_handle`.

```
extern struct FILE *fs_handle ; // file handle for
                                // "new_file.txt"
char *pbuf = "Hello World" ;
if(fputs(pbuf, fs_handle) > 0)
 printf("\n successfully written to the file");
else
 printf("\n error in writing the string");
```

FSEEK

Include

```
#include "cfileapi.h"
```

Prototype

```
int fseek(FILE *stream, long offset, int origin);
```

Description

The `fseek()` function moves the file pointer (if any) associated with `stream` to a new location that is offset a number of bytes from the origin. The next operation on the stream occurs at the new location. This API returns an error if it is called on a file that is opened in APPEND (a or ab) mode.

Argument(s)

`stream`	Pointer to the FILE structure (file on which the file pointer must be set).
`offset`	Number of bytes from origin.
`origin`	Specifies the origin from which the offset number of bytes is added to set the new file pointer position; can contain the following values: `SEEK_CUR`: Current position of file pointer. `SEEK_END`: End of file. `SEEK_SET`: Beginning of file.

Return Value(s)

If successful, `fseek()` returns 0. Otherwise, it returns a nonzero value.

Example

Use the `fseek()` function sets the file position pointer to 100 from the beginning of the file in which the handle is stored in `fs_handle`.

```
extern struct FILE *fs_handle ; // file handle for
                                // "new_file.txt"
if(fseek(fs_handle, 100, SEEK_SET) == 0)
 printf("\n successfully set the file pointer position
in the file");
else
 printf("\n error in fseek");
```

FTELL

Include
```
#include "cfileapi.h"
```

Prototype
```
long ftell(FILE *stream);
```

Description

The `ftell()` function obtains the current position of the file pointer (if any) associated with `stream`. The position of the file pointer is expressed as an offset value relative to the beginning of the stream.

Argument(s)

`stream` Pointer to the FILE structure (the file upon which the file pointer is obtained).

Return Value(s)

The `ftell()` function returns the current file position. The value returned by `ftell` may not reflect the physical byte offset for streams opened in text mode, because text mode causes carriage return-line feed (CR-LF) translation. Use `ftell` with `fseek` to return to file locations correctly. On error, `ftell()` returns `-1L`.

Example

Use `ftell()` function obtains the size of the file in which the handle is stored in `fs_handle`.

```
long lbegin ;
long lend ;
extern struct FILE *fs_handle ; // file handle for
                                //"new_file.txt"
                                //long lbegin, lend ;
```

```
if(fseek(fs_handle, 0, SEEK_SET) == 0)
 printf("\n successfully set the file pointer position
in the file");
else
 printf("\n error in fseek");

lbegin = ftell(fs_handle);

if(fseek(fs_handle, 0, SEEK_END) == 0)
 printf("\n successfully set the file pointer position
in the file");
else
 printf("\n error in fseek");

lend = ftell(fs_handle);

 printf("\n size of the file = %ld", (lend-lbegin));
```

FEOF

Include

```
#include "cfileapi.h"
```

Prototype

```
int feof(FILE *stream);
```

Description

The `feof()` routine (implemented both as a function and as a macro) determines whether the end of stream has been reached. When end of file is reached, `read` operation returns an end-of-file indicator until the stream is closed or until `fseek` is called against it.

Argument(s)

`stream`	Pointer to the FILE structure (file on which to find an EOF condition).

Return Value(s)

The `feof()` function returns a nonzero value after the first read operation that attempts to read past the end of the file. It returns 0 if the current position is not end of file. There is no error return.

Example

Use the `feof()` function verifies whether it is an `end of file` condition or not for a file whose handle is contained in `fs_handle`.

```
extern ZFS_HANDLE_t fs_handle ;
if(feof(fs_handle))
 printf("\n END Of File reached");
else
 printf("\n no End Of File is reached");
```

Appendix A. Zilog File System Data Types, Macros and Data Structures

This appendix describes the data types, macros and data structures that are used by the Zilog File System APIs.

Zilog File System Data Types

Table 4 lists the data types used by the Zilog File System. These data types are dependent on the data types employed by the Zilog Real-Time Kernel and are described in the Zilog Real-Time Kernel Reference Manual (RM0006).

Table 4. Zilog File System Data Types

Data Type	Definition
ZFS_STATUS_t	INT (integer)
ZFS_HANDLE_t	VOID * (void pointer)
FILE	VOID (void data type)

Zilog File System Macros

Table 5 lists the macros used by the Zilog File System. You can use these macros to pass values and to interpret return values within parameters or various data structures.

Table 5. Zilog File System Macros

Macro	Description

ZFS_DIR_TYPE	This macro identifies whether the entry in ZFS_FD_LIST_t structure is a directory or not. When checking for the entry type, use the following pseudocode.
	If ((~(fd_list->fd_type)) & ZFS_DIR_TYPE)
	// It is a directory.
	If((~(fd_list->fd_type)) & ZFS_FILE_TYPE)
	// It is a file.

ZFS_FILE_TYPE	This macro identifies whether the entry in the ZFS_FD_LIST_t structure is a file or not. When checking for entry type, use the following pseudocode.
	If((~(fd_list->fd_type)) & ZFS_DIR_TYPE)
	// It is a directory.
	If((~(fd_list->fd_type)) & ZFS_FILE_TYPE)
	// It is a file.

ZFS_FILE_BEGIN	This macro is used by the ZFSSeek() API to pass the original value. It seeks the beginning of the file, see ZFSSeek on page 49.
ZFS_FILE_CURRENT	This macro is used by the ZFSSeek() API to pass the original value. It seeks the current file pointer position in the file, see ZFSSeek on page 49.
ZFS_FILE_END	This macro is used by the ZFSSeek() API to pass the original value. It seeks the end of the file, see ZFSSeek on page 49.
ZFS_READ	This macro is used by the ZFSOpen() API to pass the mode of file open. This macro is used to open the file in READ mode, see ZFSOpen on page 37.
ZFS_WRITE	This macro is used by the ZFSOpen() API to pass the mode of file open. This macro is used to open the file in WRITE mode, see ZFSOpen on page 37.

Table 5. Zilog File System Macros (Continued)

Macro	Description
ZFS_APPEND	This macro is used by the ZFSOpen( ) API to pass the mode of file open. This macro is used to open the file in APPEND mode, see ZFSOpen on page 37.
ZFS_READ_WRITE	This macro is used by the ZFSOpen( ) API to pass the mode of file open. This macro is used to open the file in READ_WRITE mode, see ZFSOpen on page 37.
ZFS_MODE_ASCII	This macro is used by the ZFSOpen( ) API to pass the type of file open. This macro is used to open a file in translation mode, see ZFSOpen on page 37.
ZFS_MODE_BINARY	This macro is used by the ZFSOpen( ) API to pass the type of file open. This macro is used to open a file in no-translation mode, see ZFSOpen on page 37.

Zilog File System Data Structures

The two Zilog File System data structures, ZFS_FD_LIST_t and ZFS_VOL_PARAMS_t, are described in this section.

ZFS_FD_LIST_t

This structure is used to store file or directory attributes, such as the name, size, and the time of modification.

```
typedef struct {
  INT8    fd_name[ ZFS_MAX_FILE_NAME_SIZE + 1 ] ;
  UINT8   fd_type ;   // entry typem, DIRECTORY or
  FILE
  UINT32  fd_size;    // size of the file
  UINT8   fd_sec;     // TimeStamp-seconds
  UINT8   fd_min;     // TimeStamp-Minutes
  UINT8   fd_hrs;     // TimeStamp-Hours
```

```
  UINT8   fd_day;     // TimeStamp-Dat
  UINT8   fd_mon;     // TimeStamp-Month
  UINT8   fd_year     // TimeStamp-Year
  UINT8   fd_century; // TimeStamp-Century
} ZFS_FD_LIST_t ;
```

To decode the contents of `fd_type`, use the following pseudocode.

```
If((~(fd_list->fd_type)) & ZFS_DIR_TYPE)
 // It is a directory
If((~(fd_list->fd_type)) & ZFS_FILE_TYPE)
 // It is a file.
```

ZFS_VOL_PARAMS_t

This structure is used to store Zilog File System information such as volume name, free space, used space, and dirty space contained in the volume. Memory space is stored in bytes.

```
typedef struct {
  UINT8 vol_name[ ZFS_MAX_FILE_NAME_SIZE + 1 ] ;
  UINT8 is_valid ;  // ZFS_TRUE indicates volume is
          // valid, ZFS_FALSE indicates vol-
                  // ume is invalid.
  UINT32  vol_size ;
  UINT32  free_space ;
  UINT32  used_space ;
  UINT32  dirty_space ;
} ZFS_VOL_PARAMS_t, *PZFS_VOL_PARAMS_t ;
```

Appendix B. Zilog File System Error Codes

The Zilog File System returns error codes depending on the execution of a Zilog File System APIs. The error codes mentioned in Table 6 are valid only for Zilog File System APIs and are not applicable to C file APIs.

Table 6. Zilog File System Error Codes

Return Value	Error Codes
ZFSERR_SUCCESS	0
ZFSERR_INVALID_HANDLE	–1
ZFSERR_INVALID_ARGUMENTS	–2
ZFSERR_NOT_INITIALIZED	–3
ZFSERR_INVALID_FILEDIR_PATH	–4
ZFSERR_INVALID_OPERATION	–5
ZFSERR_DIRECTORY_NOT_EMPTY	–6
ZFSERR_INVALID_FILE_DIR_NAME	–7
ZFSERR_FILE_DIR_ALREADY_EXISTS	–8
ZFSERR_FILE_DIR_COUNT_LIMIT_REACHED	–9
ZFSERR_DIR_COUNT_LIMIT_REACHED	–10
ZFSERR_DATAMEDIA_FULL	–11
ZFSERR_INTERNAL	–12
ZFSERR_FILE_DIR_DOES_NOT_EXIST	–13
ZFSERR_FILE_DIR_IN_USE	–14
ZFSERR_INVALID_OFFSET_RANGE	–15
ZFSERR_FILE_IS_ALREADY_OPEN	–16

Table 6. Zilog File System Error Codes (Continued)

Return Value	Error Codes
ZFSERR_MAX_FILE_OPEN_COUNT_REACHED	–17
ZFSERR_DEVICE	–18
ZFSERR_INVALID_VOLUME_NAME	–19
ZFSERR_VOLUME_IS_IN_USE	–20
ZFSERR_ALREADY_SHUTDOWN	–21
ZFSERR_FS_BUSY	–22
ZFSERR_ALREADY_INITIALIZED	–23
ZFSERR_CWD_PATH_LENGTH_MORE	–24
ZFSERR_INVALID_VOLUME	–25

Customer Support

To share comments, get your technical questions answered, or report issues you may be experiencing with our products, please visit Zilog's Technical Support page at http://support.zilog.com.

To learn more about this product, find additional documentation, or to discover other facets about Zilog product offerings, please visit the Zilog Knowledge Base at http://zilog.com/kb or consider participating in the Zilog Forum at http://zilog.com/forum.

This publication is subject to replacement by a later edition. To determine whether a later edition exists, please visit the Zilog website at http://www.zilog.com.