USER'S MANUAL

TargetFFS-NAND™

NAND Flash File System

BLUNK
Microsystems

TargetFFS-NAND™

NAND Flash File System

User's Manual

Blunk Microsystems
6576 Leyland Park Drive
San Jose, CA 95120-4558
Tel: 408/323-1758
Fax: 408/323-1757
Web: www.blunkmicro.com
support@blunkmicro.com

Contents

Features 1

TargetFFS-NAND™, Blunk Microsystems' NAND Flash File System, includes the following features:

- A reliable, re-entrant file system with a full POSIX and ANSI C compliant application program interface. Use of NAND flash media for the backing store is invisible to the application layer.

- Supports dynamic creation and deletion of files, directories, and links with full read and write capability. Not a static ROM-image file system.

- Implements wear-leveling to prolong life of the flash media. Erase cycles are spread evenly across all erasable blocks. The wear count is maintained starting with the first time a flash volume is formatted. The current wear count, and other statistics, are available to applications via the vstat() call.

- Detects bad NAND blocks and performs bad block recovery without user intervention. After detection, bad blocks are neither written to nor erased.

- File system integrity is guaranteed across unexpected shutdowns. Only data written since the last synchronizing operation (fclose(), fflush(), etc.) can be lost. Closed files, directory structures, and files open for reading are never at risk.

- "Thin" driver layer for maximum ease in porting to new platforms. The driver performs only the basic operations that NAND flash devices support: read page, write page, erase block, etc. The driver is stateless. Sample drivers are provided for several chips and these are easily ported to new devices.

- Allows concurrent use of multiple volumes. Flash volumes can be permanently installed at startup, or added and deleted as removable devices come and go. Volume size is unlimited. A single volume can be implemented using multiple NAND devices.

- Supports the "self", "group", and "other" file access protections, allowing applications to restrict some operations to privileged tasks. The file system calls FsGetId(), implemented by the application, to get the running task's user and group IDs.

- Includes fast and efficient error correction code (ECC) routines. Detects up to 8 bit errors and corrects up to 4 bit errors. Hardware based ECC can be used if available.

- Optimized for fast mounts. Typical mount time for a 64MB volume is approximately one second.

- Per task current working directories. The current working directory (CWD) is specified by two 32-bit variables. TargetFFS-NAND calls two application functions to read and write the CWD state variables. If a real time operating system is used and these variables are accessed in a task specific way, each task has its own CWD.

- Garbage collection is performed to ensure minimal use of RAM for file system control information. Recycle operations, which convert dirty sectors to free sectors, may be performed in the background by calling vclean() from the idle task.

- Allows a number of flash sectors to be reserved, producing an early volume full indication. The file system will immediately exchange reserved free sectors for dirty application sectors. When combined with background recycling, this ensures there is always a pool of free sectors available, boosting file system responsiveness for user interface applications, even when the volume is full or nearly full.

- Provides fflush() and sync() for application control of file system synchronization. Supports atomic file updates using vclean() to convert dirty sectors to free and vstat() to determine the number of available free sectors before the next recycle is required.

- Supports creation of special fixed length files using creatn() for reduced file system overhead and higher write performance.

- Shipped with five sample applications: a binary search application, a power-loss recovery test, a shell that supports "cd", "ls", "mkdir", "pwd", etc, and two applications that test assorted file system calls. The shell may be extended with user commands.

- Source code is 100% ANSI C and has been tested using ANSI C compilers from ARM, Diab Data, Metrowerks, and WindRiver.

- Developed using TargetOS™, Blunk Microsystems' full-featured royalty-free real time operating system. Easily ported to other operating systems. Uses per-volume access semaphores to allow independent concurrent access to multiple volumes.

- Royalty-free site license. Includes source code, user's manual, sample applications, and one year of technical support.

Using TargetFFS-NAND 2

TargetFFS-NAND Overview

TargetFFS-NAND is a file system designed for NAND flash memory. Flash memory has qualities that are quite different from media normally used for backing store, such as hard disks, floppy disks, or magnetic tape. For best results, the architecture of a NAND flash file system must take into account the unique attributes of NAND flash memory. TargetFFS-NAND is designed from the ground up to use NAND flash memory as backing store.

The qualities of NAND flash memory include: (1) a write operation can only clear bits, bits are set only by erasing a large block, typically 8 KB or larger, (2) only a limited number of writes to the same 512 byte page can be performed between erase cycles, (3) some blocks in the NAND memory may fail, both during manufacture and normal use, (4) bit errors may occur during normal operation, and (5) blocks experience wear fatigue from program/erase cycles.

Wear fatigue results in manufacturers only guaranteeing proper operation for a finite number of program/erase cycles, typically 1 million or more. The manufacturers' guarantee applies independently to each block in the device. If one block is cycled at a higher rate than others, it will have more wear and can be expected to fail prematurely.

The design of a flash file system is very much affected by the wear leveling restriction. File systems for freely writable media typically place their control information in a fixed location and repeatedly modify this location as files and directories are created or deleted and as files expand or contract. If applied to flash, this approach would lead to uneven program/erase fatigue. TargetFFS-NAND distributes program/erase wear in an even fashion.

TargetFFS-NAND accommodates bad blocks, both those initially marked by the manufacturer and those that arise from failure of a block erase or page program operation. A block is never erased or programmed once it is determined to be bad. When TargetFFS-NAND encounters a previously unformatted volume, it searches for bad blocks. Because manufacturers fully erase all good blocks prior to shipment, any block on an unformatted volume that has a zero bit is recorded as a bad block.

Because bit errors can occur in NAND memories due to "loss of charge", TargetFFS-NAND provides ECC encoding and decoding routines that can be used by the flash driver. To support products which guarantee that no bit errors will occur below a specified number of program and erase cycles, TargetFFS-NAND passes the block wear control to the driver's read and write page routines. ECC encoding and decoding can be omitted until the wear count reaches the critical value. If hardware ECC support is provided, the driver can use that instead of the software routines provided by TargetFFS-NAND.

Because of its design, TargetFFS-NAND is robust against unexpected power-loss. When the file system's state changes, new information is written to the flash without overwriting old information; neither file system control information nor user data. Each copy of control information is given an incrementing sequence number and protected with a 32-bit cyclic redundancy check. The pages that contain control information are marked by a flag value in the page's extra bytes.

TargetFFS-NAND mounts a volume by searching for pages that contain control information. The control information that passes the cyclic redundancy check and has the highest sequence number is used as descriptive of the file system's state. The only possible difference from the state prior to a power-loss is that some pages and blocks recorded as free may have been programmed. TargetFFS-NAND scans each page recorded as free and either erases blocks or marks pages as dirty, as appropriate.

If an unexpected power loss occurs before a control write is complete, upon subsequent power-up the file system reverts to its state before the most recent change. If power loss occurs after a control write completes, the new state changes have been successfully stored and will be used when the file system reboots. In neither case are closed files, directory structures, or files open for reading at risk. Only data written since the last synchronizing operation (fclose(), fflush(), etc.) can be lost.

File system data is written to 'free' sectors. This changes the sector's marking from 'free' to 'used'. When files are deleted or overwritten, 'used' sectors become 'dirty' sectors. From time to time, depending on the size of the control information and the free sectors count, the file system selects an erase set consisting of one or more blocks, copies their 'used' sectors to another block, and then erases the blocks in the erase set. This is called a 'recycle' operation and is required to replenish the free sector list.

This explanation simplifies many details. TargetFFS-NAND incorporates clever algorithms, diligent implementation, and years of testing to provide a reliable, re-entrant file system whose use of NAND flash memory for the backing store is invisible to the application layer.

Configuration

The TargetFFS-NAND compile-time configuration parameters are contained in the files "posix.h", "stdio.h", and "flashfsp.h". The following is an example of valid settings for these parameters:

From “posix.h”:

```
/*************************************************************************/
/* Configuration                                                         */
/*************************************************************************/
#define DIROPEN_MAX     FOPEN_MAX   /* max number of open dirs */
#define PATH_MAX        FILENAME_MAX /* maximum path name length */
#define _PATH_NO_TRUNC  1           /* don't truncate path name */
#define IGNORE_CASE     0           /* no path name case sensitivity */

/*
** Cache policy - when set to 0 becomes copy back
*/
#define FFS_CACHE_WRITE_THROUGH     0

/*
```

```
** Max Number of Volumes of Each Type
*/
#define NUM_RFS_VOLS    0
#define NUM_ZFS_VOLS    0
#define NUM_FFS_VOLS    1
#define NUM_FAT_VOLS    0
#define NUM_NAND_FTLS   0

#define INC_NAND_FS     1
#define INC_NOR_FS      0
#define INC_OLD_NOR_FS  0
```

DIROPEN_MAX: The maximum number of concurrently open (via **opendir()**) directories.

PATH_MAX: The maximum number of characters in a path name, including the file name separators and terminating NULL.

_PATH_NO_TRUNC: Determines what happens if the file system is passed a path name longer than PATH_MAX. If '1', there is an error return with errno set to ENAMETOOLONG. Otherwise, the path name is truncated to length PATH_MAX.

IGNORE_CASE: If '1', file and directory name comparisons performed by the file system are case insensitive. Otherwise, case is used to differentiate file and directory names.

FFS_CACHE_WRITE_THROUGH: If '1', sectors in the RAM sector cache are written to flash as soon as they become full. Otherwise, sectors are committed to flash when either the cache is full and a new entry is needed, or the cache is flushed, such as when file system control information is written to flash.

NUM_FFS_VOLS: Maximum number of concurrently mounted volumes. Targets with only fixed media, such as soldered on chips, should leave NUM_FFS_VOLS at '1', the default value. For targets that additionally support removable flash or are using both TargetFFS-NAND and TargetFFS-NOR, setting NUM_FFS_VOLS to '2' allows simultaneous use of two flash volumes.

NUM_RFS_VOLS, NUM_ZFS_VOLS, and NUM_FAT_VOLS apply to TargetRFS, TargetZFS, and TargetFAT. They should be left as '0' unless those file systems are being used with TargetFFS-NAND. NUM_NAND_FTL applies to TargetFTL-NAND and should be left as 0.

INC_NAND_FS: Must be '1' for TargetFFS-NAND.

INC_NOR_FS: Must be '0' unless you are using TargetFFS-NAND with TargetFFS-NOR.

INC_OLD_NOR_FS: Should be left as '0'.

From “stdio.h”:

```
/***********************************************************************/
/* Configuration                                                       */
/***********************************************************************/
#define FOPEN_MAX    64
```

```
#define FILENAME_MAX 63
#define L_tmpnam     14
#define TMP_MAX      10000
```

FOPEN_MAX: The maximum number of concurrently open files (the value in "posix.h" must match the value in "stdio.h").

FILENAME_MAX: The maximum length of a file name string (the value in "posix.h" must match the value in "stdio.h"). _PATH_NO_TRUNC determines whether longer filenames cause an error return or are truncated

L_tmpnam: The maximum length of a filename generated by tmpnam(). Must be >= 5 + LOG(TMP_MAX).

TMP_MAX: The number of unique filenames that tmpnam() will generate.

From “flashfsp.h”:

```
/***********************************************************************/
/* Configuration                                                       */
/***********************************************************************/
#define BACKWARD_COMPATIBLE     1
#define FFS_DEBUG               0

/*
** Extra free space (not available to the app) for faster performance.
** To be used in conjunction with vclean()
*/
#define EXTRA_FREE              0   /* in bytes */

/*
** If 1, changes the machine endianess for the control information
** when it is written to / read from flash.
*/
#define FFS_SWAP_ENDIAN         0
```

BACKWARD_COMPATIBLE: Set to '1' for compatibility with releases prior to release 2004.0. If '0', one unnecessary byte is eliminated from the control information written to flash.

EXTRA_FREE: Determines how much flash memory is reserved for background garbage collection, as discussed in the Section "EXTRA_FREE" below.

FFS_SWAP_ENDIAN: If '0', the control information is written to flash in the natural byte order of the host CPU: either big or little endian. Set to '1' to force the control information to be written in the opposite order. This is useful if the flash is removable and the volume will be mounted on both big and little endian machines.

To use TargetFFS-NAND with TargetOS, the function **FfsModule()** must be added to the application’s Module List. Flash driver modules can be added to the Module List as well, for automatic installation during system startup. To ensure TargetFFS-NAND is initialized before flash drivers are installed, the TargetFFS-NAND module function must be listed ahead of any flash driver modules. See the TargetOS User's Manual for more information about the Module List.

Also, the necessary libraries must be linked with the application. The TargetFFS-NAND library “flash.a” is always required. To use the Standard C <stdio.h> API, include either “runlib.a” or “runlibf.a” depending on whether the integer-only or floating-point routines, respectively, are needed. To include the POSIX file-related API, include “posix.a”.

To make a TargetFFS-NAND volume accessible, so it can be formatted, mounted, etc., call **FsNandAddVol()**. This can be done in the driver's initialization function. **FsNandAddVol()** can be called once at startup to permanently install a flash volume or **FsNandAddVol()** and **FfsDel-Vol()** can be called repeatedly during operation as removable flash volumes come and go. See Chapter 4, “Flash Driver Interface”, for more information.

Volume Information

TargetOS file systems use **vstat()** to provide file system specific information for a named volume. **vstat()** writes its output to a union whose address is passed as a parameter. This union contains a structure for each TargetOS file system. The TargetFFS-NAND specific structure is defined in “posix.h” and reproduced below:

```
typedef struct
{
  ui32 vol_type;
  ui32 sect_size;       /* sector size in bytes */
  ui32 used_sects;      /* number of used sectors */
  ui32 num_sects;       /* number of sectors */
  ui32 avail_sects;     /* number of available sectors */
  ui32 sects_2recycle;  /* number of available sectors before recycle */
  ui32 flash_type;      /* FFS_NAND or FFS_NOR */
  ui32 wear_count;      /* maximum wear count */
} vstat_ffs;
```

For TargetFFS-NAND volumes, ***vol_type*** is set to FFS_VOL, defined in “posix.h”. ***sect_size*** is the number of bytes in a TargetFFS-NAND sector, the minimum increment of new flash memory that is assigned to a file when it grows. Given a 512 byte driver page size, the sector size depends on the total volume size as follows:

Volume Size	Sector Size
< 32MB	512
From 32MB to less than 64MB	1024
From 64MB to less than 128MB	2048
…	…

num_sects is the total number of sectors minus the sectors in the maximum number of potentially bad blocks, as specified in **FsNandAddVol()**. ***used_sects*** is the number of sectors currently in use, for both and file system control data and user data. ***avail_sects*** is the number of sectors available for user data. ***sects_2recycle*** is the number of free sectors that can be consumed before the next recycle operation is triggered.

flash_type is used to distinguish between TargetFFS-NAND and TargetFFS-NOR volumes. It is set to FFS_NAND (defined in "posix.h") by TargetFFS-NAND.

wear_count is the number of times the first erasable block in the volume has been programmed and erased. The wear count of each block is recorded. Wear-leveling is done to ensure that no block's wear count exceeds the others by more than a small amount. The wear counts are set to zero when a flash volume is initially formatted and are remembered across subsequent format calls.

To ensure reliability, a volume's wear count should be monitored to ensure it does not exceed the number of program/erase cycles guaranteed by the manufacturer of the flash memory device. Operation past this limit is not recommended. Since this number is typically above 1 million cycles, TargetFFS-NAND volumes can be used for years without replacement.

File Access Protection

TargetFFS-NAND supports the “self”, “group”, and “other” file access protections using the application defined function **FsGetId()**. File protection applies to both directories and regular files. **FsGetId()** is called whenever a file created, to assign the user and group IDs, and whenever a file is opened, to check the user and group IDs, along with the protection modes, that are associated with the file. Its prototype is (uid_t and gid_t are unsigned short integers):

```
void FsGetId(uid_t *uid, gid_t *gid);
```

Applications can implement **FsGetId()** using static variables that are only changed when a privileged task accesses or creates a file. An example of such an implementation is:

```
static uid_t UserID, GroupID;

/***********************************************************************/
/*     FsGetId: Get process user and group ID                          */
/*                                                                     */
/*      Inputs: uid = place to store user ID                           */
/*              gid = place to store group ID                          */
/*                                                                     */
/***********************************************************************/
void FsGetId(uid_t *uid, gid_t *gid)
{
  *uid = UserID;
  *gid = GroupID;
}
```

Alternately, the user and group IDs can be dynamically associated with each task using the task registers that many RTOS’s support. The example below shows an implementation using TargetOS:

```
#define ID_REG  0
```

```
/***********************************************************************/
/*      FsGetId: Get process user and group ID                         */
/*                                                                     */
/*       Inputs: uid = place to store user ID                          */
/*               gid = place to store group ID                         */
/*                                                                     */
/***********************************************************************/
void FsGetId(uid_t *uid, gid_t *gid)
{
  ui32 id = taskGetReg(RunningTask, ID_REG);

  *uid = (uid_t)id;
  *gid = id >> 16;
}
```

FsGetId() is prototyped in “posix.h”, along with **FsSetId()**, an application function which may be implemented to initialize or modify the user and group IDs of an application or task. **FsSetId()** is a convenient way for applications to set a task's privilege level. TargetFFS-NAND does not call **FsSetId()**. An implementation for TargetOS is:

```
/***********************************************************************/
/*      FsSetId: Set process user and group ID                         */
/*                                                                     */
/*       Inputs: uid = user ID                                         */
/*               gid = group ID                                        */
/*                                                                     */
/***********************************************************************/
void FsSetId(uid_t uid, gid_t gid)
{
#if 1
  ui32 id = (gid << 16) | uid;

  taskSetReg(RunningTask, ID_REG, id);
#else
  UserID = uid;
  GroupID = gid;
#endif
}
```

Per Task CWD

TargetFFS-NAND supports per task current working directories. The current working directory (CWD) state information is recorded in two 32-bit values. TargetFFS-NAND calls **FsReadCWD()** to read these values when it needs to access the CWD, such as to resolve a relative path. **FsSaveCWD()** is called to save the CWD state information when it has been changed, such as by an application call to **chdir()**. The prototypes for these functions (in "posix.h") are:

```
void FsReadCWD(ui32 *word1, ui32 *word2);
void FsSaveCWD(ui32 word1, ui32 word2);
```

Both **FsReadCWD()** and **FsSaveCWD()** are implemented by the application. They are not implemented by TargetFFS-NAND. If the application is not using a real-time operating system or does not need per task current working directories, it can save the two CWD state information variables in global variables. It then has a single CWD directory. An example of such an implementation is:

```
static ui32 Word1, Word2;

/***********************************************************************/
/*   FsSaveCWD: Save current working directory state information       */
/*                                                                     */
/*      Inputs: word1 = 1 of 2 words to save                           */
/*              word2 = 2 of 2 words to save                           */
/*                                                                     */
/***********************************************************************/
void FsSaveCWD(ui32 word1, ui32 word2)
{
  Word1 = word1;
  Word2 = word2;
}

/***********************************************************************/
/*   FsReadCWD: Read current working directory state information       */
/*                                                                     */
/*     Outputs: word1 = 1 of 2 words to retrieve                       */
/*              word2 = 2 of 2 words to retrieve                       */
/*                                                                     */
/***********************************************************************/
void FsReadCWD(ui32 *word1, ui32 *word2)
{
  *word1 = Word1;
  *word2 = Word2;
}
```

If the CWD state information variables are saved in a task unique way, the application can have per task current working directories. An example of such an implementation using TargetOS is:

```
#define CWD_WD1         1
#define CWD_WD2         2

/***********************************************************************/
/*   FsSaveCWD: Save per-task current working directory state          */
/*                                                                     */
/*      Inputs: word1 = 1 of 2 words to save                           */
/*              word2 = 2 of 2 words to save                           */
/*                                                                     */
/***********************************************************************/
void FsSaveCWD(ui32 word1, ui32 word2)
{
  taskSetReg(RunningTask, CWD_WD1, word1);
  taskSetReg(RunningTask, CWD_WD2, word2);
}
```

```
/***********************************************************************/
/*   FsReadCWD: Read per-task current working directory state         */
/*                                                                     */
/*     Outputs: word1 = 1 of 2 words to retrieve                       */
/*              word2 = 2 of 2 words to retrieve                       */
/*                                                                     */
/***********************************************************************/
void FsReadCWD(ui32 *word1, ui32 *word2)
{
  *word1 = taskGetReg(RunningTask, CWD_WD1);
  *word2 = taskGetReg(RunningTask, CWD_WD2);
}
```

Before a task that will access the file system makes its first access, it should initialize its CWD state variables to zero (i.e. "FsSaveCWD(0, 0);"). If the state variables have a random value, it could cause a bus error or memory corruption within TargetFFS-NAND. When the variables are zero, the CWD is the root directory ("/").

Similarly, before a task is deleted that has used **chdir()** to change its CWD from the root directory, it should change its CWD back to the root directory. TargetFFS-NAND prevents directories that are the CWD of any task from being deleted. If a task is deleted without changing its CWD to the root directory, the directory that was its CWD can never be removed.

Background Garbage Collection

When files are deleted or overwritten, dirty sectors are generated. Normally, these dirty sectors are not reclaimed until the next file operation that consumes a free sector. At that time, the file system determines if the free sector count has reached a low level threshold and performs a recycle if the free sector count is too low.

A recycle consists of selecting an erase set that consists of one or more blocks, copying the 'used' sectors to another block, and then erasing the blocks in the erase set. Recycles are necessary to keep from running out of free sectors. When a recycle operation occurs, it implements a synchronization point that saves the current state of the file system to flash.

Recycles can be done ahead of time by calling **vclean()**. **vclean()** takes one parameter, an ASCII string containing the name of the volume to be recycled. It returns -1 if there is an error, 0 if no additional dirty sectors can be freed, and 1 if more sectors can be cleaned. To avoid 'hogging' access to the file system, each call to **vclean()** performs at most a single recycle. To fully clean a volume, **vclean()** should be called until it no longer returns 1, as in this example:

```
/***********************************************************************/
/*     cleaner: Reclaims dirty sectors in 'background'. If the flash   */
/*              driver is guaranteed not to block (i.e. uses polling), */
/*             then this vclean() call may be moved to the idle task.  */
/*                                                                     */
/***********************************************************************/
static void cleaner(void)
{
  for (;;)
```

```
    {
      while (vclean("flash") > 0) ;
      taskSleep(2 * OsTicksPerSec);
    }
  }
```

EXTRA_FREE

Unlike a hard disk file system, dirty flash sectors are not immediately available for further use, but must be recovered through a recycle operation. This can cause apparent sluggishness in the following situations: 1) writing over an existing file in a volume that is full or 2) writing a new file immediately after deleting a file from a volume that is full.

To improve responsiveness in these situations, TargetFFS-NAND allows definition of an amount of flash memory that is withheld from the user. The macro EXTRA_FREE in "flashfsp.h" specifies the number of bytes that the file system reserves for itself. This definition is used to produce an early volume full indication. The file system reserves the extra sectors for itself, but will immediately exchange its reserved free sectors for the application's dirty sectors.

This has the effect of immediately transforming the application's dirty sectors into free sectors and is especially effective when used with background garbage collection. As long as **vclean()** is called often enough between successive demands for free sectors in a full or nearly full file system, there will always be a standby pool of free sectors which will boost the responsiveness of the file system.

Synchronization

When data is written to a TargetFFS-NAND volume, it is initially stored in a RAM sector cache. Various operations flush the cached data to flash. This is called synchronization. When a volume is re-mounted after an unexpected power loss, it reverts to its state at the last synchronization point prior to the power loss. Directory structures, closed files, and files open for reading are never at risk, but data written after the last synchronization can be lost.

The following service calls synchronize a volume (**fopen()** and **open()** only cause synchronization if a file is created or truncated):

chmod()	**chown()**	**close()**	**creat()**	**creatn()**	**fclose()**
fflush()	**fopen()**	**freopen()**	**ftruncate()**	**link()**	**mkdir()**
open()	**remove()**	**rename()**	**rmdir()**	**sync()**	**truncate()**
unlink()	**unmount()**	**utime()**			

In addition, recycle operations cause synchronization. When an application writes to a file and the data crosses a sector boundary, a free sector is allocated. This can trigger a recycle and its associated synchronization. Also, **vclean()** causes synchronization if there are dirty sectors to reclaim and it performs a recycle.

In addition to flushing the sector cache, synchronization writes file system control information (file names, directory structures, etc.) to flash. It is sometimes desirable to avoid this overhead.

An example is transferring many files in a manufacturing environment. In this case, if the transfer is interrupted for any reason, it may not be important that files copied prior to the interruption are saved to the flash. It is usually more important to avoid the overhead of writing file system control information to flash after each file is created.

To temporary disable the synchronization caused by most of the listed service calls, TargetFFS-NAND provides the routines **disable_sync()** and **enable_sync()**. These calls have a single parameter; the volume name as an ASCII string. They return -1 if there is an error, otherwise, 0. The synchronization caused by **unmount()** or a write operation that triggers a recycle can not be avoided.

Atomic Updates

Some applications require atomic file updates. This means that either all writes to a file succeed or, if there is a power loss before the last update, the file reverts to its initial state, with no possibility that the updates be left half-completed. Although this is a hard problem for a file system to solve generally without adding unacceptable overhead, it can be easily solved at the application layer above the file system.

One method uses a separate log file that fully describes the desired modifications in idempotent steps (steps that can be repeated without changing the final outcome). After the steps in the log file are executed one by one, the log file is deleted. If the application finds, at power-up, that a log file exists, it simply repeats the listed steps and deletes the log file after the last step. This ensures the listed modifications succeed in an atomic fashion.

TargetFFS-NAND does not provide a general atomic update solution. This is because any write that crosses a sector boundary can trigger a recycle and its associated synchronization. Even a single write is not guaranteed to be atomic if the data spans multiple sectors. If a recycle is followed by a power loss, the file will be modified up to the boundary of the sector whose allocation triggered the recycle. After that point, the file will contain old data.

TargetFFS-NAND provides limited support for atomic updates via **vclean()** and **vstat()**. The method is as follows. Before making a file update, call **vclean()** repeatedly until it returns 0, indicating that no additional dirty sectors can be reclaimed. Next, read the ***sects_2recycle*** field of the structure output by **vstat()**. This indicates the number of available free sectors before the next recycle is required. If the number of sectors modified by the update does not exceed this value, then the update can be done atomically, even if it consists of multiple write operations. After the last write, call **sync()** or **fflush()** to save the update to flash.

If multiple tasks are potentially writing to the file system, some means of synchronization is required to ensure that free sectors intended for the atomic update are not consumed for other purposes.

RAM Requirements

A small component of the TargetFFS-NAND memory use is statically allocated variables. The majority of the used memory is dynamically allocated when a volume is mounted. Three data structures use most of the memory. They are listed below, with equations for calculating their size:

- Cache – Buffers accesses to flash. Size = Sector Size * (FOPEN_MAX + 1) + 76 * FOPEN_MAX.
- FILE Array – File Control Blocks. Size = FOPEN_MAX * 64.
- Sectors Array – Records sector assignment and state. Size = 4 * number of sectors.

The number and size of sectors in a volume is determined by the volume size. The table below shows the minimum RAM requirements for an assortment of volume sizes with FOPEN_MAX set to 64 and FILENAME_MAX set to 63.

Volume Size:	8-15 MB	16-31 MB	32-63 MB	64-127 MB	128-255 MB
Sector Size	512 B		1 KB	2 KB	4 KB
# Sectors	16384-30720	32768-63488	32768-64512	32768-65024	32768-65280
Cache	~37 KB		~68 KB	~135 KB	~265 KB
FILE Array	4 KB				
Sectors Array	64-120 KB	128-248 KB	128-252 KB	128-254 KB	128-255 KB
Min RAM	~120-170 KB	~180-300 KB	~210-330 KB	~280-410 KB	~410-530 KB

In addition to the structures listed above which are fixed by the amount of flash memory assigned to the volume, there is another data structure whose size depends on the number of files created by the application. Information about every file on the volume, including name, mode, and directory links, is stored in RAM tables. Each table contains multiple entries.

When a file is created and all existing table entries are full, a new table is dynamically allocated. When a file is deleted and the number of used entries falls below a threshold, entries on the table containing the fewest entries are copied elsewhere and the newly empty table is freed. The amount of table space used by each file is 2 * (FILENAME_MAX + 28). It is important to understand that the file system's RAM requirements will increase as the number of files increases.

File System API 3

Introduction

The TargetOS file system API includes the file-related routines from both Standard C and POSIX. Either the posix.h or stdio.h header file must be included to use the routines, as indicated in the pages that follow which document each routine. The supplied functions support all common file operations.

In addition to the Standard C and POSIX routines, the API includes nine non-standard routines: creatn(), disable_sync(), enable_sync(), format(), mount(), sync(), unmount(), vclean(), and vstat(). These routines were added to support operations not typically allowed by application programs in the “big computer” heritage shared by Standard C and POSIX.

Except for rename(), the Standard C API is a subset of the POSIX API in regard to functionality. The function names and parameters are not the same, but equivalent operations are performed. (sync() is grouped with the POSIX API and is equivalent to the Standard C fflush().)

This chapter is shared by all TargetOS file systems. The descriptions that follow contain a key in their top right-hand corner indicating which file systems support the specified routine. Appendix D presents the same information in tabular form. The keys use the following mapping:

FFS - TargetFFS-NAND and NOR flash file systems
FAT - TargetFAT DOS-compatible file system
RFS - TargetRFS RAM based file system
ZFS - TargetZFS compressed read-only file system

FAT • FFS • RFS • ZFS

access()

PROTOTYPE

```
#include <posix.h>

int access(const char *path, int amode);
```

DESCRIPTION

access() is used to test a file for accessibility. ***path*** must be a valid path name, absolute if starting with '/', otherwise relative to the current working directory. ***amode*** determines the access permissions to be checked. It is a bit-wise OR of the following values:

• **R_OK**	Test if file is readable.
• **W_OK**	Test if file is writable.
• **X_OK**	Test if file is executable.
• **F_OK**	Test if file exists.

If successful, **access()** returns 0. Otherwise, errno is set and -1 is returned.

ERROR CODES

EACCES	Access error (no execute permission for directory in path prefix or the requested access test failed).
EFAULT	***path*** equals NULL.
ENAMETOOLONG	Path name length exceeds **PATH_MAX** and **_PATH_NO_TRUNC** is TRUE.
ENOENT	The file specified by ***path*** was not found.
ENOTDIR	One of the prefix names in ***path*** is not a directory.
ENXIO	The volume specified by the path root name is not mounted.

EXAMPLE

```
/*-------------------------------------------------------------*/
/* Check if the current working directory can be read.         */
/*-------------------------------------------------------------*/
if (access(“.”, R_OK))
  printf(“Can’t call opendir() on this directory!");
```

FAT • FFS • RFS • ZFS

chdir()

PROTOTYPE

```
#include <posix.h>

int chdir(const char *path);
```

DESCRIPTION

chdir() is used to change the current working directory to the directory specified by ***path***. ***path*** must be a valid directory path name, absolute if starting with '/', otherwise relative to the current working directory. If an error occurs, **chdir()** does not change the current working directory.

If successful, **chdir()** returns 0. Otherwise, errno is set and -1 is returned.

ERROR CODES

EACCES	No execute permission for directory in path prefix.
EFAULT	***path*** equals NULL.
ENAMETOOLONG	Length exceeds **PATH_MAX** and **_PATH_NO_TRUNC** is TRUE.
ENOENT	The directory specified by ***path*** was not found.
ENOTDIR	One of the prefix names in ***path*** is not a directory.
ENXIO	The volume specified by the path root name is not mounted.

EXAMPLE

```
/*-------------------------------------------------------------*/
/* Change current working directory to specified path.         */
/*-------------------------------------------------------------*/
if (chdir(path))
{
  /*-----------------------------------------------------------*/
  /* chdir() failed (current working directory is unchanged).  */
  /*-----------------------------------------------------------*/
  perror("error trying to change directory!");
  return –1;
}
```

FFS • RFS • ZFS

chmod()

PROTOTYPE

```
#include <posix.h>

int chmod(const char *path, mode_t mode);
```

DESCRIPTION

chmod() is used to change the access protection modes of the file specified by ***path***. ***path*** must be a valid path name, absolute if starting with '/', otherwise relative to the current working directory. ***mode*** sets the file's permission bits and should be a bit-wise OR of the following values:

• **S_ISUID**	Set user ID on execution (ignored).
• **S_ISGID**	Set group ID on execution (ignored).
• **S_IRUSR**	Allow read permission to owner.
• **S_IWUSR**	Allow write permission to owner.
• **S_IXUSR**	Allow execute permission to owner.
• **S_IRGRP**	Allow read permission to group.
• **S_IWGRP**	Allow write permission to group.
• **S_IXGRP**	Allow execute permission to group.
• **S_IROTH**	Allow read permission to everyone.
• **S_IWOTH**	Allow write permission to everyone.
• **S_IXOTH**	Allow execute permission to everyone.

If successful, **chmod()** returns 0. Otherwise, errno is set and -1 is returned.

ERROR CODES

EACCES	No execute permission for directory in path prefix.
EFAULT	***path*** equals NULL.
ENAMETOOLONG	Length exceeds **PATH_MAX** and **_PATH_NO_TRUNC** is TRUE.
ENOENT	The file specified by ***path*** was not found.
ENOTDIR	A component of the path prefix is not a directory.
EPERM	User ID determined by **FsGetId()** does not match file's owner.
ENXIO	The volume specified by the path root name is not mounted.

EXAMPLE

```
/*-----------------------------------------------------------------*/
```

```
/* Allow all permissions to self and read permission to group.   */
/*---------------------------------------------------------------*/
if (chmod(path, S_RUSR | S_WUSR | S_XUSR | S_RGRP))
{
  perror("error trying to change file access mode!");
  return –1;
}
```

FFS • RFS • ZFS

chown()

PROTOTYPE

```
#include <posix.h>

int chown(const char *path, uid_t owner, gid_t group);
```

DESCRIPTION

chown() is used to set a file's owner and group IDs. ***path*** must be a valid path name, absolute if starting with '/', otherwise relative to the current working directory. ***owner*** and ***group*** are the new owner and group IDs, respectively.

If successful, **chown()** returns 0. Otherwise, errno is set and -1 is returned.

ERROR CODES

EACCES	No execute permission for directory in path prefix.
EFAULT	***path*** equals NULL.
ENAMETOOLONG	Length exceeds **PATH_MAX** and **_PATH_NO_TRUNC** is TRUE.
ENOENT	The file specified by ***path*** was not found.
ENOTDIR	A component of the path prefix is not a directory.
EPERM	User ID determined by **FsGetId()** does not match file's owner.
ENXIO	The volume specified by the path root name is not mounted.

EXAMPLE

```
/*-------------------------------------------------------------*/
/* Change file's group ID to the privileged group.             */
/*-------------------------------------------------------------*/
if (chown(path, AppID, PrivGp))
{
  perror("error trying to change file owner!");
  return –1;
}
```

FAT • FFS • RFS • ZFS

clearerr()

PROTOTYPE

```
#include <stdio.h>

void clearerr(FILE* stream);
```

DESCRIPTION

clearerr() is used to clear the error and end-of-file indicators for the I/O stream identified by ***stream***.

ERROR CODES

EINVAL	***stream*** is invalid.

EXAMPLE

```
/*-----------------------------------------------------------------*/
/* Clear the error for stream_a.                                   */
/*-----------------------------------------------------------------*/
clearerr(stream_a);
```

FAT • FFS • RFS • ZFS

close()

PROTOTYPE

```
#include <posix.h>

int close(int fid);
```

DESCRIPTION

close() is used to free the file descriptor specified by ***fid***. Any buffered output is written to the stream. If ***fid*** is the last descriptor that refers to the associated file, any buffered input is discarded and the file mode is reset. After **close()** returns, the value of ***fid*** may not be used to reference the closed I/O stream.

If successful, **close()** returns 0. Otherwise, errno is set and -1 is returned.

ERROR CODES

EBADF	The file descriptor ***fid*** is invalid.

EXAMPLE

```
/*-------------------------------------------------------------*/
/* Close file and check if successful.                         */
/*-------------------------------------------------------------*/
if (close(file_descriptor))
  perror("error trying to close file!");
```

FAT • FFS • RFS • ZFS

closedir()

PROTOTYPE

```
#include <posix.h>

int closedir(DIR *dirp);
```

DESCRIPTION

closedir() is used to close a directory. ***dirp*** must be a directory handle returned by **opendir()**. After **closedir()** returns, the value of ***dirp*** may not be used to reference the closed directory.

If successful, **closedir()** returns 0. Otherwise, errno is set and -1 is returned.

ERROR CODES

EBADF	The directory handle ***dirp*** is invalid.

EXAMPLE

```
/*-------------------------------------------------------------*/
/* Close directory and return if error.                        */
/*-------------------------------------------------------------*/
if (closedir(curr_dir))
{
  perror("error in closing directory!");
  return –1;
}
```

FAT • FFS • RFS

creat()

PROTOTYPE

```
#include <posix.h>

int creat(const char *path, mode_t mode);
```

DESCRIPTION

creat() is used to create and open a file. ***path*** must be a valid path name, absolute if starting with '/', otherwise relative to the current working directory. ***mode*** sets the file's permission bits and should be a bit-wise OR of the following values:

• **S_IRUSR**	Allow read permission to owner.
• **S_IWUSR**	Allow write permission to owner.
• **S_IXUSR**	Allow execute permission to owner.
• **S_IRGRP**	Allow read permission to group.
• **S_IWGRP**	Allow write permission to group.
• **S_IXGRP**	Allow execute permission to group.
• **S_IROTH**	Allow read permission to everyone.
• **S_IWOTH**	Allow write permission to everyone.
• **S_IXOTH**	Allow execute permission to everyone.

creat(path, mode) is equivalent to **open(path, O_WRONLY | O_CREAT | O_TRUNC, mode)**.

If successful, **creat()** returns a file descriptor, a small integer used by other I/O functions to reference the file. Otherwise, errno is set and -1 is returned.

ERROR CODES

EACCES	Either no execute permission for path directory, no write permission for parent directory, or no read or write permission for specified file.
EFAULT	***path*** equals NULL.
EINVAL	The specified file name is invalid.
EISDIR	The specified file name is a directory.
ENAMETOOLONG	Length exceeds **PATH_MAX** and **_PATH_NO_TRUNC** is TRUE.
EMFILE	No file control block is free. FOPEN_MAX files are currently open.
ENOENT	A parent directory not found.
ENOMEM	Unable to allocate memory to create file entry.
ENOSPC	No space is left on volume to create file.

ENOTDIR	A component of the path prefix is not a directory.
ENXIO	The volume specified by the path root name is not mounted.

EXAMPLE

```
fid = creat("foo", 0666);
if (fid == -1)
  perror("error creating the file!");
```

creatn()

PROTOTYPE

```
#include <posix.h>

int creatn(const char *path, mode_t mode, size_t size);
```

DESCRIPTION

creatn() is used to create a special file whose size is fixed. The file length is ***size*** bytes rounded up to a multiple of the underlying volume's sector size. In other respects, the behavior is similar to **creat()**. See **creat()** for information about ***path***, ***size***, and the possible error codes. Because the file's size is fixed, **creatn()** allows various file system dependent optimizations. It is guaranteed that writes to files created by **creatn()** will not trigger a recycle operation.

When **creatn()** is used with TargetFFS-NOR, there is one restriction. The application can seek over the file and write data multiple times, but the application must ensure there is no attempt to change a previously written '0' to a '1'.

When **creatn()** is used with TargetFFS-NAND, there are three restrictions: 1) the length of each write must be an integer multiple of a full sector size, 2) writes must begin at a sector boundary, and 3) file data can only be written once, over-writes are not allowed.

If successful, **creatn()** returns a file descriptor, a small integer used by other I/O functions to reference the file. Otherwise, errno is set and -1 is returned.

EXAMPLE

```
/*-------------------------------------------------------------------*/
/* Create a special file of length 2048.                             */
/*-------------------------------------------------------------------*/
fid = creatn(fname, 0666, 2048);
if (fid == -1)
{
  perror("error creating file!");
  return -1;
}
```

FFS

disable_sync()

PROTOTYPE

```
#include <posix.h>

int disable_sync(const char *path);
```

DESCRIPTION

disable_sync() is used to disable the file system synchronizations associated with the service calls listed in "Synchronization" in Chapter 2. ***path*** is the root name of the affected volume. If a listed routine, other than **unmount()**, is called while synchronizations are disabled, the synchronization is deferred and will be performed later when synchronization is re-enabled. Synchronizations caused by **unmount()** or a recycle are never deferred. File system synchronization is enabled by default.

disable_sync() returns zero if successful, -1 on error.

ERROR CODES

EFAULT	***path*** equals NULL.
ENAMETOOLONG	Length exceeds **PATH_MAX** and **_PATH_NO_TRUNC** is TRUE.
ENOENT	No volume matching the specified name has been mounted.

EXAMPLE

```
/*-------------------------------------------------------------------*/
/* Disable volume synchronizations during multiple FTP transfers.    */
/*-------------------------------------------------------------------*/
if (disable_sync("flash"))
  perror("error disabling synchronization");
```

FAT • FFS • RFS • ZFS

dup()

PROTOTYPE

```
#include <posix.h>

int dup(int fid);
```

DESCRIPTION

dup() is used to allocate and initialize a second descriptor that refers to the same open file as ***fid***. The lowest numbered available descriptor is used. The new descriptor should be freed using **close()** when no longer needed.

"**dup(fid)"** is equivalent to "**fcntl(fid, F_DUPFD, 0)**;".

If successful, **dup()** returns a new file descriptor. Otherwise, errno is set and -1 is returned.

ERROR CODES

EBADF	The file descriptor ***fid*** is invalid.
EMFILE	No file control block is free. FOPEN_MAX files are currently open.

EXAMPLE

```
/*-------------------------------------------------------------------*/
/* Get file ID for stdout and use duplicate to write "Hello\n".      */
/*-------------------------------------------------------------------*/
fid = fileno(stdout);
fid2 = dup(fid);
write(fid2, "Hello\n", 6);
close(fid2);
```

FAT • FFS • RFS • ZFS

dup2()

PROTOTYPE

```
#include <posix.h>

int dup2(int fid, int fid2);
```

DESCRIPTION

dup2() is used to initialize ***fid2*** as a second descriptor that refers to the same open file as ***fid***. If ***fid2*** equals ***fid***, **dup2()** just returns ***fid***. Otherwise, if ***fid2*** refers to an open file, that file is closed. The file control block associated with ***fid2*** is then reused.

If successful, **dup2()** returns the descriptor of an open file. That descriptor should be free using **close()** when it is no longer needed. If there is an error, errno is set and -1 is returned.

ERROR CODES

EBADF	The file descriptor ***fid*** is invalid.
EMFILE	No file control block is free. FOPEN_MAX files are currently open.

EXAMPLE

```
/*-----------------------------------------------------------------*/
/* Get file ID for stdout and use duplicate to write "Hello\n".      */
/*-----------------------------------------------------------------*/
fid = fileno(stdout);
fid2 = dup2(fid);
write(fid2, "Hello\n", 6);
close(fid2);
```

FFS

enable_sync()

PROTOTYPE

```
#include <posix.h>

int enable_sync(const char *path);
```

DESCRIPTION

enable_sync() is used to re-enable the file system synchronizations associated with the service calls listed in "Synchronization" in Chapter 2. ***path*** is the root name of the affected volume. File system synchronization is enabled by default. If a synchronization was deferred by a prior call to **disable_sync()**, then the sector cache is flushed and control information is written to flash before **enable_sync()** returns. Synchronizations caused by **unmount()** or a recycle are never deferred.

enable_sync() returns zero if successful, -1 on error.

ERROR CODES

EFAULT	***path*** equals NULL.
ENAMETOOLONG	Length exceeds **PATH_MAX** and **_PATH_NO_TRUNC** is TRUE.
ENOENT	No volume matching the specified name has been mounted.

EXAMPLE

```
/*-----------------------------------------------------------------*/
/* Re-enable file system control writes after multiple FTP transfers.*/
/*-----------------------------------------------------------------*/
if (enable_sync("flash"))
  perror("error enabling synchronization");
```

FAT • FFS • RFS • ZFS

fclose()

PROTOTYPE

```
#include <stdio.h>

int fclose(FILE *stream);
```

DESCRIPTION

fclose() is used to close the I/O stream identified by ***stream***. Any buffered output is written to the stream. Any buffered input is discarded. If ***stream*** is the last reference to the file, the file control block is freed and the file mode is reset. After **fclose()** returns, the value of ***stream*** may not be used to reference the closed I/O stream.

If successful, **fclose()** returns 0. Otherwise, errno is set and EOF is returned.

ERROR CODES

EBADF	***stream*** is invalid.
EOF	The operation failed.

EXAMPLE

```
/*-------------------------------------------------------------*/
/* Close file associated with stream_a and return if error.    */
/*-------------------------------------------------------------*/
if (fclose(stream_a))
{
  perror("error trying to close file!");
  return –1;
}
```

FAT • FFS • RFS • ZFS

fcntl()

PROTOTYPE

```
#include <posix.h>

int fcntl(int fid, int cmd, ...);
```

DESCRIPTION

fcntl() is used to perform the command specified by ***cmd*** to the open file specified by ***fid***. The supported commands are F_DUPFD, F_GETFL, and F_SETFL.

When used with F_DUPFD, **fcntl()** takes three parameters. It allocates a duplicate descriptor that refers to the same open file as ***fid*** using the lowest numbered available descriptor that is greater than or equal to the file number specified as the third parameter. The duplicate descriptor is reserved and should be freed using **close()** when no longer needed. If successful, **fcntl()** returns the duplicate file descriptor. Otherwise, errno is set and -1 is returned.

When used with F_GETFL, **fcntl()** takes two parameters and returns the descriptor's flags, as set by **open()**. The possible flags are: O_APPEND, O_ASYNC, O_NONBLOCK, O_RDONLY, O_RDWR, O_WRONLY. If an error occurs, errno is set and -1 is returned.

When used with F_SETFL, **fcntl()** takes three parameters. The descriptor's flags are set to the value supplied as the third parameter. The only flags that may be set are: O_APPEND, O_ASYNC, and O_NONBLOCK. Other flag values are ignored. If successful, **fcntl()** returns 0. Otherwise, errno is set and -1 is returned.

Though, for compatibility, O_ASYNC and O_NONBLOCK can be set and read back, these flags do not apply to TargetFFS and have no effect on its operation.

ERROR CODES

EBADF	The file descriptor ***fid*** or the F_DUPFD file number is invalid.
EINVAL	The command specified by ***cmd*** is not F_DUPFD or the third parameter is negative or greater than or equal to FOPEN_MAX.
EMFILE	No file control block is free. FOPEN_MAX files are currently open.

EXAMPLE

```
/*-------------------------------------------------------------------*/
/* Get file ID for stdout and use duplicate to write "Hello\n".      */
/*-------------------------------------------------------------------*/
fid = fileno(stdout);
```

```
fid2 = fcntl(fid, F_DUPFD, 0);
write(fid2, "Hello\n", 6);
close(fid2);
```

FAT • FFS • RFS • ZFS

fdopen()

PROTOTYPE

```
#include <posix.h>

FILE *fdopen(int fid, const char *mode);
```

DESCRIPTION

fdopen() is used to associate a Standard C stream with the POSIX file descriptor ***fid***. It does the opposite of **fileno()**. ***mode*** is ignored. If successful, **fdopen()** returns a stream handle. The stream's error and end-of-file indicators are cleared, and the file position indicator is set to the offset for the file descriptor.

If there is an error, errno is set and **fdopen()** returns NULL.

ERROR CODES

EBADF	The file descriptor ***fid*** is invalid.

EXAMPLE

```
int fn;
FILE *fp;

/*------------------------------------------------------------------*/
/* Open file for reading.                                           */
/*------------------------------------------------------------------*/
fn = open("/flash/sample.txt", O_RDONLY);
if (fn == -1)
{
  perror("error opening the file");
  return -1;
}

/*--------------------------------------------------------------*/
/* Convert file descriptor to stream handle.                    */
/*--------------------------------------------------------------*/
fp = fdopen(fn, 0);
if (fp == NULL)
  perror("fdopen() error");
```

FAT • FFS • RFS • ZFS

feof()

PROTOTYPE

```
#include <stdio.h>

int feof(FILE *stream);
```

DESCRIPTION

feof() is used to test the end-of-file indicator of the open I/O stream identified by ***stream***. It returns zero if the end-of-file indicator for the stream is not set, nonzero if it is set.

If there is an error, errno is set and **feof()** returns -1.

ERROR CODES

EBADF	***stream*** is invalid.

EXAMPLE

```
stream_a = fopen("/flash/temp.txt", "r");

/*------------------------------------------------------------*/
/* Output the file, a character at a time.                    */
/*------------------------------------------------------------*/
while (feof(stream_a) == 0)
  putchar(fgetc(stream_a));
```

FAT • FFS • RFS • ZFS

ferror()

PROTOTYPE

```
#include <stdio.h>

int ferror(FILE *stream);
```

DESCRIPTION

ferror() is used to test the error indicator for the open I/O stream identified by ***stream***. It returns nonzero if and only if either the stream's error indicator is set or ***stream*** is invalid.

If there is an error, errno is set and **ferror()** returns -1.

ERROR CODES

EBADF	***stream*** is invalid.

EXAMPLE

```
/*-------------------------------------------------------------*/
/* Check if any error occurred.                                */
/*-------------------------------------------------------------*/
if (ferror(stream_a))
  perror();
```

FAT • FFS • RFS

fflush()

PROTOTYPE

```
#include <stdio.h>

int fflush(FILE *stream);
```

DESCRIPTION

fflush() is used to flush buffered data, writing it to the underlying storage media. If ***stream*** is NULL, the buffered data for every open file is flushed. Otherwise, just data for the specified file is flushed.

If successful, **fflush()** returns 0. Otherwise, errno is set and EOF is returned.

ERROR CODES

EBADF	***stream*** is invalid.

EXAMPLE

```
/*-----------------------------------------------------------------*/
/* Flush buffered data to the storage media.                       */
/*-----------------------------------------------------------------*/
if (fflush(stream))
  return –1;
```

FAT • FFS • RFS • ZFS

fgetc()

PROTOTYPE

```
#include <stdio.h>

int fgetc(FILE *stream);
```

DESCRIPTION

fgetc() is used to read one character from the I/O stream identified by ***stream***. The character is read from the current file position, advancing the file position indicator, if defined.

If successful, **fgetc()** returns the character as an unsigned character converted to type “int”. Otherwise, errno is set and EOF is returned.

ERROR CODES

EACCES	File is not open in read mode.
EBADF	***stream*** is invalid.
EIO	An input error occurred.
EISDIR	***stream*** refers to a directory.

EXAMPLE

```
/*-------------------------------------------------------------*/
/* Open the file and write it to stdout.                       */
/*-------------------------------------------------------------*/
stream_a = fopen("/flash/temp.txt", "r");
while (feof(stream_a) == 0)
  putchar(fgetc(stream_a));
```

FAT • FFS • RFS • ZFS

fgetpos()

PROTOTYPE

```
#include <stdio.h>

int fgetpos(FILE *stream, fpos_t *pos);
```

DESCRIPTION

fgetpos() is used to get the file position indicator for a stream. The position indicator for the file identified by ***stream*** is stored at ***pos***. This value can only be used in later calls to **fsetpos()**. Application programs cannot interpret the fpos_t type.

If successful, **fgetpos()** returns zero. Otherwise, errno is set and a nonzero value is returned.

ERROR CODES

EBADF	***stream*** is invalid.

EXAMPLE

```
fpos_t pos;

/*-----------------------------------------------------------------*/
/* Get current position for the stream.                            */
/*-----------------------------------------------------------------*/
if (fgetpos(stream, &pos))
{
  perror("error getting position indicator for file");
  return -1;
}
```

FAT • FFS • RFS • ZFS

fgets()

PROTOTYPE

```
#include <stdio.h>

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

DESCRIPTION

fgets() is used to read a string from the open I/O stream identified by ***stream***. ***s*** points to a receive buffer. Characters are copied to the receive buffer until either ***n*** – 1 characters are read, a newline character is read, or end-of-file is reached. The string is terminated with a NULL character.

If successful, **fgets()** returns ***s***. Otherwise, errno is set and NULL is returned.

ERROR CODES

EACCES	File is not open in read mode.
EBADF	***stream*** is invalid.
EIO	An input error occurred.
EISDIR	***stream*** refers to a directory.

EXAMPLE

```
char s[20];
FILE *stream;

if ((stream = fopen("/flash/temp.txt", "r")) == NULL)
{
  perror("error opening the file!");
  return -1;
}

/*----------------------------------------------------------------*/
/* Read the first 20 chars from stream.                           */
/*----------------------------------------------------------------*/
if (fgets(s, 21, stream) == NULL)
{
  perror("error reading from file");
  return -1;
}
```

FAT • FFS • RFS • ZFS

fileno()

PROTOTYPE

```
#include <posix.h>

int fileno(FILE *stream);
```

DESCRIPTION

fileno() is used to convert a Standard C stream handle to a POSIX file descriptor. It does the opposite of **fdopen()**. ***stream*** must be a valid handle for an open stream.

If successful, **fileno()** returns the corresponding file descriptor. Otherwise, errno is set and -1 is returned.

ERROR CODES

EBADF	***stream*** is invalid.

EXAMPLE

```
/*-------------------------------------------------------------------*/
/* Get file ID for stdout and use duplicate to write "Hello\n".      */
/*-------------------------------------------------------------------*/
fid = fileno(stdout);
fid2 = fcntl(fid, F_DUPFD, 0);
write(fid2, "Hello\n", 6);
```

FAT • FFS • RFS • ZFS

fopen()

PROTOTYPE

```
#include <stdio.h>

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

DESCRIPTION

fopen() is used to open a file and associate it with a stream. ***filename*** must be a valid pathname. Pathnames are absolute if they start with '/', otherwise they are relative to the current working directory. ***mode*** must be one of the following:

• "r", "rb"	open an existing file for reading.
• "w", "wb"	create a new file or open and truncate an existing file for writing.
• "a", "ab"	create a new file or open an existing file for writing. Each write is appended to the end of the file.
• "r+", "r+b", "rb+"	open an existing file for reading and writing.
• "w+", "w+b", "wb+"	create a new file or open and truncate an existing file for reading and writing.
• "a+", "a+b", "ab+"	create a new file or open an existing file for reading and writing. Each write is appended to the end of the file.

No distinction is made between text files and binary files. Files created by **fopen()** have read, write, and execute permission bits set only for the owner. The new file's owner and group IDs are determined by a call to **FsGetId()**.

If successful, **fopen()** returns pointer to the stream. Otherwise, errno is set and NULL is returned.

ERROR CODES

EACCES	Either no execute permission for path directory, no write permission for parent directory, or no read or write permission for specified file.
EINVAL	Either ***filename*** or ***mode*** are NULL, or the file name is invalid.
EISDIR	***filename*** refers to a directory and ***mode*** requires write access.
ENAMETOOLONG	Length exceeds **PATH_MAX** and **_PATH_NO_TRUNC** is TRUE.
EMFILE	No file control block is free. FOPEN_MAX files are currently open.
ENOENT	The specified parent directory was not found.
ENOSPC	No space is left on volume to create file.
ENOTDIR	A component of the path prefix is not a directory.
ENXIO	The volume specified by the path root name is not mounted.

EXAMPLE

```
FILE *stream_a;

/*-----------------------------------------------------------------*/
/* Open file for reading.                                          */
/*-----------------------------------------------------------------*/
if ((stream_a = fopen("/flash/temp.txt", "r")) == NULL)
{
  perror("error opening the file!");
  return -1;
}
```

FAT • FFS • RFS

format()

PROTOTYPE

```
#include <posix.h>

int format(char *path);
```

DESCRIPTION

format() is used to format the volume specified by **path**. **path** must match the root name of an installed volume. The specified volume can be either mounted or unmounted. If the volume is mounted, **format()** closes any open files. Formatting neither mounts an unmounted volume nor unmounts a mounted volume.

format() returns zero if successful, -1 on error after setting errno to the appropriate error code.

ERROR CODES

EFAULT	***path*** equals NULL.
ENAMETOOLONG	Length exceeds **PATH_MAX** and **_PATH_NO_TRUNC** is TRUE.
ENOENT	No volume matching the specified name has been installed.

EXAMPLE

```
/*-------------------------------------------------------------------*/
/* Format the flash file system and check for success.             */
/*-------------------------------------------------------------------*/
if (format("flash"))
  printf("format(\"flash\") failed!\n");
```

FAT • FFS • RFS

fprintf()

PROTOTYPE

```
#include <stdio.h>

int fprintf(FILE *stream, const char *format, …);
```

DESCRIPTION

fprintf() is used to print formatted text to the I/O stream identified by ***stream***. ***format*** and any following arguments are interpreted according to Standard C’s output format string conventions, as documented in Appendix B.

If successful, **fprintf()** returns the number of characters written to ***stream***. Otherwise, errno is set and a negative value is returned.

ERROR CODES

EACCES	File is not open in append or write mode.
EBADF	***stream*** is invalid.
EIO	An output error occurred.
ENOSPC	Volume is full.

EXAMPLE

```
FILE *stream_a;

/*-----------------------------------------------------------------*/
/* Open file for writing.                                          */
/*-----------------------------------------------------------------*/
if ((stream_a = fopen("/flash/temp.txt", "w")) == NULL)
  return -1;

/*-----------------------------------------------------------------*/
/* Write simple message to stream.                                 */
/*-----------------------------------------------------------------*/
if (fprintf(stream_a, "Hello World!\n") == EOF)
  return -1;
```

FAT • FFS • RFS

fputc()

PROTOTYPE

```
#include <stdio.h>

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

DESCRIPTION

fputc() is used to write one character to the I/O stream identified by ***stream***. The character ***ch*** is converted to type “unsigned char” and written at the current file position, advancing the file position indicator, if defined.

If successful, **fputc()** returns the character written. Otherwise, errno is set and EOF is returned.

ERROR CODES

EACCES	File is not open in append or write mode.
EBADF	***stream*** is invalid.
EIO	An output error occurred.
ENOSPC	Volume is full.

EXAMPLE

```
FILE *stream_a;

/*-----------------------------------------------------------------*/
/* Open file for appended writes.                                  */
/*-----------------------------------------------------------------*/
if ((stream_a = fopen("/flash/temp.txt", "a")) == NULL)
  return -1;

/*-----------------------------------------------------------------*/
/* Write one letter to the end of stream.                          */
/*-----------------------------------------------------------------*/
if (fputc(‘H’, stream_a) != ‘H’)
  return -1;
```

FAT • FFS • RFS

fputs()

PROTOTYPE

```
#include <stdio.h>

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

DESCRIPTION

fputs() is used to write the character string specified by **string** to the I/O stream identified by **stream**. The ending NULL character is not written.

If successful, **fputs()** returns a positive value. Otherwise, errno is set and EOF is returned.

ERROR CODES

EACCES	File is not open in append or write mode.
EINVAL	***string*** equals NULL.
EBADF	***stream*** is invalid.
EIO	An output error occurred.
ENOSPC	Volume is full.

EXAMPLE

```
FILE *stream_a;

/*-----------------------------------------------------------------*/
/* Open file for writing.                                          */
/*-----------------------------------------------------------------*/
if ((stream_a = fopen("/flash/temp.txt", "w")) == NULL)
  return -1;

/*-----------------------------------------------------------------*/
/* Write string to stream.                                         */
/*-----------------------------------------------------------------*/
if (fputs("Hello World!\n", stream_a) == EOF)
  return -1;
```

FAT • FFS • RFS • ZFS

fread()

PROTOTYPE

```
#include <stdio.h>

size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
```

DESCRIPTION

fread() is used to read an array of elements from the I/O stream identified by ***stream***. **fread()** writes consecutive elements to the buffer starting at ***ptr*** until either an error occurs or ***nmemb*** elements have been read. ***size*** is the number of bytes in a single element.

If ***size*** or ***nmemb*** equal 0, **fread()** returns 0. If nothing is read due to an error, errno is set and EOF is returned. Otherwise, **fread()** returns the number of elements read. That is the number of bytes read divided by ***size***.

ERROR CODES

EACCES	File is not open in read mode.
EINVAL	***ptr*** equals NULL.
EBADF	***stream*** is invalid.
EIO	An input error occurred.
EISDIR	***stream*** refers to a directory.

EXAMPLE

```
FILE *stream_a;
char buffer[20];

/*-----------------------------------------------------------------*/
/* Open file for reading.                                          */
/*-----------------------------------------------------------------*/
if ((stream_a = fopen("/flash/temp.txt", "r")) == NULL)
  return -1;

/*-----------------------------------------------------------------*/
/* Read the first 20 characters from stream.                       */
/*-----------------------------------------------------------------*/
if (fread(buffer, 1, 20, stream_a) != 20)
  return -1;
```

FAT • FFS • RFS • ZFS

freopen()

PROTOTYPE

```
#include <stdio.h>

FILE *freopen(const char *filename,  const char *mode, FILE *stream);
```

DESCRIPTION

freopen() is used to close the file identified by ***stream*** and open the file specified by ***filename*** based on ***mode*** (which is interpreted as by **fopen()**). ***filename*** must be a valid pathname, absolute if starting with '/' and relative to the current working directory otherwise. The following:

```
stream = freopen(filename, mode, stream)
```

is equivalent to:

```
fclose(stream);
stream = fopen(filename, mode);
```

If successful, **freopen()** returns ***stream***. Otherwise, errno is set and NULL is returned.

ERROR CODES

EACCES	Either no execute permission for path directory, no write permission for parent directory, or no read or write permission for specified file.
EBADF	***stream*** is invalid.
EINVAL	Either ***mode*** equals NULL or the file name is invalid.
EISDIR	***filename*** refers to a directory and ***mode*** requires write access.
ENAMETOOLONG	Length exceeds **PATH_MAX** and **_PATH_NO_TRUNC** is TRUE.
ENOENT	The specified parent directory was not found.
ENOSPC	No space is left on volume to create file.
ENOTDIR	A component of the path prefix is not a directory.
ENXIO	The volume specified by the path root name is not mounted.

EXAMPLE

```
FILE *stream_a;

/*-----------------------------------------------------------------*/
/* Open file for reading.                                          */
```

```
/*----------------------------------------------------------------*/
if ((stream_a = fopen("/flash/temp.txt", "r")) == NULL)
{
  perror("error opening the file!");
  return -1;
}

/*----------------------------------------------------------------*/
/* Reopen the same handle for a different file.                   */
/*----------------------------------------------------------------*/
if (freopen("/flash/new_temp.txt", "r+", stream_a) == NULL)
{
  perror("error reopening file");
  return -1;
```

FAT • FFS • RFS • ZFS

fscanf()

PROTOTYPE

```
#include <stdio.h>

int fscanf(FILE *stream, const char *format, …);
```

DESCRIPTION

fscanf() is used to receive formatted input. Input from the I/O stream identified by ***stream*** is interpreted according to Standard C's input format string conventions. The values, if any, are stored in the additional arguments. The Standard C input format string specification included in Appendix B.

If no input assignments are made before either end-of-file is reached or a conversion error occurs, errno is set and **fscanf()** returns EOF. Otherwise, **fscanf()** returns the number of input assignments.

ERROR CODES

EACCES	File is not open in read mode.
EBADF	***stream*** is invalid.
EIO	An input error occurred.
EISDIR	***stream*** refers to a directory.

EXAMPLE

```
FILE *stream_a;
int i;

/*-----------------------------------------------------------------*/
/* Open file for reading.                                          */
/*-----------------------------------------------------------------*/
if ((stream_a = fopen("/flash/temp.txt", "r")) == NULL)
  return -1;

/*-----------------------------------------------------------------*/
/* Read an integer from stream.                                    */
/*-----------------------------------------------------------------*/
if (fscanf(stream_a, "%d", &i) == EOF)
  return -1;
```

FAT • FFS • RFS • ZFS

fseek()

PROTOTYPE

```
#include <stdio.h>

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

DESCRIPTION

fseek() sets the file position indicator for the I/O stream identified by ***stream***. The new position is ***offset*** number of bytes from the position determined by ***whence***. ***whence*** must be one of the following:

- **SEEK_SET** – ***offset*** is the new absolute offset (from the beginning of file).
- **SEEK_CUR** – ***offset*** is relative to the current file position.
- **SEEK_END** – ***offset*** is relative to the end of the file.

Except for the special fixed length files, **fseek()** may be used to move the file position indicator past the end of existing data. If a write is performed while the position indicator is beyond the end of the file, the file is expanded. The gap between the position indicator and the previous end is filled with NULL characters.

If successful, **fseek()** returns zero and clears the effects of **ungetc()**. Otherwise, errno is set and -1 is returned.

ERROR CODES

EACCES	Seeking past end and file is not open in append or write mode.
EBADF	***stream*** is invalid.
EINVAL	***whence*** is invalid or attempted to seek past end of fixed length file.
ENOSPC	No space is left on volume to expand file.

EXAMPLE

```
/*-----------------------------------------------------------------*/
/* Output the first 12 characters of file in reverse order.        */
/*-----------------------------------------------------------------*/
for (i = 12; i >= 0; --i)
{
  if (fseek(stream_a, i, SEEK_SET))
    return -1;
  putchar(getc(stream_a));
}
```

FAT • FFS • RFS • ZFS

fsetpos()

PROTOTYPE

```
#include <stdio.h>

int fsetpos(FILE *stream, const fpos_t *pos);
```

DESCRIPTION

fsetpos() is used to set the file position for the I/O stream identified by ***stream***. The position is set to the value stored in ***pos***, which must have been obtained by a prior call to **fgetpos()**.

If successful, **fsetpos()** clears the stream's end-of-file indicator, removes the effects of any **ungetc()** calls, and returns zero. Otherwise, errno is set and a nonzero value is returned.

ERROR CODES

EBADF	***stream*** is invalid.

EXAMPLE

```
/*-----------------------------------------------------------------*/
/* Get current position for the stream.                            */
/*-----------------------------------------------------------------*/
if (fgetpos(stream_a, &pos))
{
  perror("error getting position indicator for file");
  return -1;
}

/*-----------------------------------------------------------------*/
/* Set the position for stream_b to that of stream_a.              */
/*-----------------------------------------------------------------*/
if (fsetpos(stream_b, &pos))
{
  perror("error setting position indicator for file");
  return -1;
}
```

FAT • FFS • RFS • ZFS

fstat()

PROTOTYPE

```
#include <posix.h>

int fstat(int fid, struct stat *buf);
```

DESCRIPTION

fstat() is used to obtain information about a file. ***fid*** must be a valid identifier for an open file. The status information is written to the struct stat structure pointed to by ***buf***. The stat structure definition is contained in “posix.h” and reproduced below:

```
struct stat
{
  mode_t  st_mode;   /* file mode */
  ino_t   st_ino;    /* file serial number */
  dev_t   st_dev;    /* ID of device containing this file */
  nlink_t st_nlink;  /* number of links */
  uid_t   st_uid;    /* user ID of file's owner */
  gid_t   st_gid;    /* group ID of file's owner */
  off_t   st_size;   /* the file size in bytes */
  time_t  st_atime;  /* time of last access */
  time_t  st_mtime;  /* time of last data modification */
  time_t  st_ctime;  /* time of last status change */
};
```

If successful, **fstat()** returns 0. Otherwise, errno is set and -1 is returned.

ERROR CODES

EBADF	***fid*** is invalid.
EFAULT	***buf*** equals NULL.

EXAMPLE

```
struct stat buf;

if (fstat(fid, &buf) == 0)
{
  printf("File access time:   %s", ctime(&buf.st_atime));
  printf("File modified time: %s", ctime(&buf.st_mtime));
  printf("File mode:          %d\n", buf.st_mode);
  printf("File serial number: %d\n", buf.st_ino);
  printf("File num links:     %d\n", buf.st_nlink);
```

```
    printf("File user ID:       %u\n", buf.st_uid);
    printf("File group ID:      %u\n", buf.st_gid);
    printf("File size (byte):   %u\n", buf.st_size);
```

FAT • FFS • RFS • ZFS

ftell()

PROTOTYPE

```
#include <stdio.h>

long ftell(FILE *stream);
```

DESCRIPTION

ftell() is used to get the current file position indicator for the I/O stream identified by ***stream***. If successful, **ftell()** returns the value of the file position indicator as the number of bytes from the beginning of the file. Otherwise, -1 is returned.

ERROR CODES

EBADF	***stream*** is invalid.

EXAMPLE

```
long pos;
FILE *stream_a;

/*-----------------------------------------------------------------*/
/* Open file for reading.                                          */
/*-----------------------------------------------------------------*/
if ((stream_a = fopen("/flash/temp.txt", "r")) == NULL)
{
  perror("error opening the file!");
  return -1;
}

/*-----------------------------------------------------------------*/
/* Get current position for the stream.                            */
/*-----------------------------------------------------------------*/
pos = ftell(stream_a);
if (pos == -1L)
{
  perror("error getting position indicator for file");
  return -1;
}
```

FAT • FFS • RFS

ftruncate()

PROTOTYPE

```
#include <posix.h>

int ftruncate(int fid, off_t length);
```

DESCRIPTION

ftruncate() is used to set the length of the regular file identified by ***fid*** to ***length*** characters. If ***length*** is greater than the file's current length, the file is extended by appending '\0' characters. Otherwise, the file is truncated to ***length*** characters. The file's position indicator is set to the end of the file.

If successful, **ftruncate()** returns 0. Otherwise, errno is set and -1 is returned.

ERROR CODES

EACCES	File is not open in append or write mode.
EBADF	***fid*** is not a valid descriptor for a file open for writing.
EINVAL	***length*** is negative.
EIO	An output error occurred.
ENOSPC	No space is left on volume to write file data.

EXAMPLE

```
/*-----------------------------------------------------------------*/
/* Shorten the file to half its length.                            */
/*-----------------------------------------------------------------*/
length = lseek(fid2, 0, SEEK_END);
if (ftruncate(fid2, length / 2))
{
  perror("error trying to truncate file!");
  return -1;
}
```

FAT • FFS • RFS

fwrite()

PROTOTYPE

```
#include <stdio.h>

size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
```

DESCRIPTION

fwrite() is used to write an array of elements to the I/O stream identified by ***stream***. **fwrite()** reads consecutive elements from the buffer starting at ***ptr*** until either an error occurs or ***nmemb*** elements have been written. ***size*** is the number of bytes in a single element.

If ***size*** or ***nmemb*** equal 0, **fwrite()** returns 0. If nothing is written due to an error, errno is set and EOF is returned. Otherwise, **fwrite()** returns the number of elements written. That is the number of bytes written divided by ***size***.

ERROR CODES

EACCES	File is not open in append or write mode.
EBADF	***stream*** is invalid.
EIO	An output error occurred.
EINVAL	***ptr*** equals NULL.
ENOSPC	No space is left on volume to expand file.

EXAMPLE

```
FILE *stream_a;

/*-------------------------------------------------------------------*/
/* Open file for writing.                                            */
/*-------------------------------------------------------------------*/
stream_a = fopen("/flash/a/b/file1.txt", "w");
if (stream_a == NULL)
  return;

/*-------------------------------------------------------------------*/
/* Write greeting to stream_a.                                       */
/*-------------------------------------------------------------------*/
if (fwrite(“Hello world!\n”, 1, 13, stream_a) != 13)
  return -1;
```

FAT • FFS • RFS • ZFS

getc()

PROTOTYPE

```
#include <stdio.h>

int getc(FILE *stream);
```

DESCRIPTION

getc() is used to read one character from the I/O stream identified by ***stream***. The character is read from the current file position, advancing the file position indicator, if defined.

If successful, **getc()** returns the character as an unsigned character converted to type “int”. Otherwise, errno is set and EOF is returned.

ERROR CODES

EACCES	File is not open in read mode.
EBADF	***stream*** is invalid.
EIO	An input error occurred.
EISDIR	***stream*** refers to a directory.

EXAMPLE

```
/*-------------------------------------------------------------*/
/* Open file and copy it to stdout.                            */
/*-------------------------------------------------------------*/
stream_a = fopen("/flash/temp.txt", "r");
while (feof(stream_a) == 0)
  printf("%c", getc(stream_a));
```

FAT • FFS • RFS • ZFS

getchar()

PROTOTYPE

```
#include <stdio.h>

int getchar(void);
```

DESCRIPTION

getchar() is used to read one character from the I/O stream identified by stdin. The character is read from the current file position, advancing the file position indicator, if defined. The function is equivalent to **fgetc(stdin)**.

If successful, **getchar()** returns the character as an unsigned character converted to type “int”. Otherwise, errno is set and EOF is returned.

ERROR CODES

EACCES	stdin is not open in read mode.
EBADF	stdin is invalid.
EIO	An input error occurred.
EISDIR	stdin refers to a directory.

EXAMPLE

```
/*------------------------------------------------------------*/
/* Echo stdin to stdout until ‘q’ is entered.                 */
/*------------------------------------------------------------*/
do
{
  ch = getchar();
  if (ch == EOF)
    perror("error reading from input!");
  putchar(ch);
} while (ch != ‘q’);
```

FAT • FFS • RFS • ZFS

getcwd()

PROTOTYPE

```
#include <posix.h>

char *getcwd(char *buf, size_t size);
```

DESCRIPTION

getcwd() is used to get the path name of the current working directory. If ***buf*** is not NULL, then ***size*** is the size of the buffer it points to. The absolute path name of the current working directory is copied to this buffer.

If ***buf*** is NULL, then ***size*** is ignored and **getcwd()** calls **malloc()** to allocate a buffer of suitable size before copying the absolute path name of the current working directory to it. This buffer must freed, by calling **free()**, after it is no longer needed by the application.

If successful, **getcwd()** returns a pointer to the absolute path name of the current working directory. Otherwise, errno is set and NULL is returned.

ERROR CODES

EACCES	Either no execute permission for directory in path prefix or no read permission for parent directory.
EINVAL	***size*** is zero and ***buf*** is not NULL.
ENOMEM	Unable to allocate memory buffer for path name.
ERANGE	***buf*** is not NULL and ***size*** is negative or less than the length of the path name, including the terminating '\0'.

EXAMPLE

```
/*-----------------------------------------------------------------*/
/* If the call to get the current dir fails, return error.         */
/*-----------------------------------------------------------------*/
cwd = getcwd(NULL, 0);
if (cwd == NULL)
  return -1;

/*-----------------------------------------------------------------*/
/* Output directory path name and free the path buffer.            */
/*-----------------------------------------------------------------*/
printf("%s\n", cwd);
free(cwd);
```

FAT • FFS • RFS • ZFS

gets()

PROTOTYPE

```
#include <stdio.h>

char *gets(char *s);
```

DESCRIPTION

gets() is used to read a string from the I/O stream identified by stdin. ***s*** points to a receive buffer. Characters are copied to the receive buffer until either a newline character is read or end-of-file is reached. The string is terminated with a NULL character.

Because no check is made that the receive buffer is big enough to hold the number of characters read, **gets()** is dangerous. Use **fgets()** instead.

If successful, **gets()** returns ***s***. Otherwise, errno is set and NULL is returned.

ERROR CODES

EACCES	File is not open in read mode.
EBADF	stdin is invalid.
EINVAL	***s*** equals NULL.
EIO	An input error occurred.
EISDIR	stdin refers to a directory.

EXAMPLE

```
char s[HUGE_BUFFER];

/*-----------------------------------------------------------------*/
/* Read string from stdin.                                         */
/*-----------------------------------------------------------------*/
if (gets(s) == NULL)
{
  perror("error reading from standard input!");
  return -1;
}
```

FAT • FFS • RFS • ZFS

isatty()

PROTOTYPE

```
#include <posix.h>

int isatty(int fid);
```

DESCRIPTION

isatty() is used to determine whether a file descriptor is associated with a terminal device.

If ***fid*** describes a terminal, **isatty()** returns 1. Otherwise, errno is set and 0 is returned.

ERROR CODES

ENOTTY	***fid*** is not associated with a terminal device.

EXAMPLE

```
/*-------------------------------------------------------------------*/
/* If device is a terminal, ring its bell.                           */
/*-------------------------------------------------------------------*/
if (isatty(fid2))
  write(fid2, "\a", 1);
```

FFS • RFS

link()

PROTOTYPE

```
#include <posix.h>

int link(const char *existing, const char *new);
```

DESCRIPTION

link() is used to create a new link to the file or directory specified by ***existing***, which must be a valid path name. A new link is created using **new**, which must be a valid path name on the same file system. Pathnames are absolute if they start with '/', otherwise they are relative to the current working directory.

If successful, **link()** returns zero. Otherwise, errno is set and -1 is returned.

ERROR CODES

EACCES	No execute permission for directory in path prefix or no write permission for parent directory.
EEXIST	***new*** specifies an existing file.
EFAULT	Either ***existing*** or ***new*** equals NULL.
EINVAL	Either ***existing*** or ***new*** are invalid path names or ***existing*** is parent directory of ***new***.
ENAMETOOLONG	Length exceeds **PATH_MAX** and **_PATH_NO_TRUNC** is TRUE.
ENOENT	The file or directory specified by ***existing*** was not found.
ENOSPC	No space is left on volume to create link.
ENOTDIR	A component of one of the path prefixes is not a directory.
EPERM	User ID determined by **FsGetId()** does not match file's owner.
EXDEV	***new*** is not on the same file system as ***existing***.
ENXIO	The volume specified by the path root name is not mounted.

EXAMPLE

```
/*-----------------------------------------------------------------*/
/* Link path2 to path1.                                            */
/*-----------------------------------------------------------------*/
if (link(path1, path2))
  return –1;
```

FAT • FFS • RFS • ZFS

lseek()

PROTOTYPE

```
#include <posix.h>

off_t lseek(int fid, off_t offset, int whence);
```

DESCRIPTION

lseek() is used to set the file position indicator for the open file specified by ***fid***. The requested position is the position indicated by ***whence*** plus ***offset*** number of bytes. If no longer at EOF, the file's end-of-file indicator is cleared. ***whence*** must be one of the following:

- **SEEK_SET** – the reference position is the beginning of the file.
- **SEEK_CUR** – the reference position is the current file position.
- **SEEK_END** – the reference position is the end of the file.

Except for the special fixed length files, **lseek()** may be used to move the file position indicator past the end of existing data. If a write is performed while the position indicator is beyond the end of the file, the file is expanded. The gap between the position indicator and the previous end is filled with NULL characters.

If successful, **lseek()** returns the new file position. Otherwise, errno is set and (off_t)-1 is returned.

ERROR CODES

EACCES	Seeking past end and file is not open in append or write mode.
EBADF	***fid*** is invalid.
EINVAL	***whence*** is invalid or attempted to seek past end of fixed length file.
ENOSPC	No space is left on volume to expand file.

EXAMPLE

```
/*-----------------------------------------------------------------*/
/* Implement binary search algorithm.                              */
/*-----------------------------------------------------------------*/
low = 0; high = RAND_MAX;
for (;;)
{
  /*---------------------------------------------------------------*/
  /* Seek to half-way point.                                       */
  /*---------------------------------------------------------------*/
```

```
    pos = (low + high) >> 1;
    sc = lseek(fid, pos * sizeof(Record), SEEK_SET);
    if (sc == (off_t)-1)
      SysFatalError(errno);

    /*---------------------------------------------------------------*/
    /* Read record at requested offset.                              */
    /*---------------------------------------------------------------*/
    rc = read(fid, &record, sizeof(Record));
    if (rc == -1)
      SysFatalError(errno);

    /*---------------------------------------------------------------*/
    /* Update high or low pointer or break, depending on comparison. */
    /*---------------------------------------------------------------*/
    if (key < record.key)
      high = pos - 1;
    else if (key > record.key)
      low = pos + 1;
    else
      break;
  }
```

FAT • FFS • RFS

mkdir()

PROTOTYPE

```
#include <posix.h>

int mkdir(const char *path, mode_t mode);
```

DESCRIPTION

mkdir() is used to create a directory. ***path*** must be a valid directory path name, absolute if starting with '/', otherwise relative to the current working directory. The new directory is initially empty. ***mode*** sets the directory's permission bits and should be a bit-wise OR of the following values:

• **S_IRUSR**	Allow read permission to owner.
• **S_IWUSR**	Allow write permission to owner.
• **S_IXUSR**	Allow execute permission to owner.
• **S_IRGRP**	Allow read permission to group.
• **S_IWGRP**	Allow write permission to group.
• **S_IXGRP**	Allow execute permission to group.
• **S_IROTH**	Allow read permission to everyone.
• **S_IWOTH**	Allow write permission to everyone.
• **S_IXOTH**	Allow execute permission to everyone.

If successful, **mkdir()** returns 0. Otherwise, errno is set and -1 is returned.

ERROR CODES

EACCES	Either no execute permission for directory in path prefix or no write permission for parent directory.
EEXIST	The named directory already exists.
EFAULT	***path*** is not a valid pointer.
EINVAL	The directory name is invalid.
ENAMETOOLONG	Length exceeds **PATH_MAX** and **_PATH_NO_TRUNC** is TRUE.
ENOENT	The parent directory specified by ***path*** was not found.
ENOMEM	Unable to allocate memory for control information.
ENOSPC	No space is left on volume to create directory.
ENOTDIR	A component of the path prefix is not a directory.
ENXIO	The volume specified by the path root name is not mounted.

EXAMPLE

```
/*---------------------------------------------------------------*/
/* Create a directory to hold the object code.                   */
/*---------------------------------------------------------------*/
if (mkdir(“obj”, S_IRUSR | S_IWUSR | S_IXUSR))
  perror("error trying to make directory!");
```

FAT • FFS • RFS • ZFS

mount()

PROTOTYPE

```
#include <posix.h>

int mount(char *name);
```

DESCRIPTION

mount() is used to mount the volume specified by **name**. After a volume is mounted, its files can be accessed by applications. Before a volume is mounted, its files are inaccessible. **name** must match the root name of an installed volume.

Before it can be mounted, a volume must be installed by one of the TargetOS file systems. Volumes are installed using a file system-dependent call. For instance, flash file volumes are installed using **FlashAddVol()**. Typically, a volume is installed by its device driver during driver initialization.

A volume that is installed but not mounted uses a minimum of system resources. Mounting the volume makes it fully ready to be used by applications. This typically requires allocating memory and perhaps additional resources.

If successful, **mount()** returns 0. Otherwise, errno is set and -1 is returned.

ERROR CODES

EEXIST	A volume with the same name is already mounted.
EFAULT	***name*** equals NULL.
ENAMETOOLONG	Length exceeds **PATH_MAX** and **_PATH_NO_TRUNC** is TRUE.
ENOENT	No volume matching the specified name has been installed.

EXAMPLE

```
/*-------------------------------------------------------------------*/
/* Mount the flash file system and check for success.              */
/*-------------------------------------------------------------------*/
if (mount("flash"))
  printf("mount(\”flash\”) failed!\n");
```

FAT • FFS • RFS • ZFS

open()

PROTOTYPE

```
#include <posix.h>

int open(const char *path, int oflag, …);
```

DESCRIPTION

open() is used to open a file. ***path*** must be a valid path name, absolute if starting with '/', otherwise relative to the current working directory. ***oflag*** indicates the file mode and must contain exactly one of the following:

• **O_RDONLY**	Open for reading only.
• **O_WRONLY**	Open for writing only.
• **O_RDWR**	Open for reading and writing.

It may also contain any of the following OR'd together with the above:

• **O_APPEND**	Set the file offset to the end of file before writing to file.
• **O_CREAT**	Allow the file to be created. **Requires** a third parameter that is the same as **creat()**'s mode.
• **O_EXCL**	Fail if the file already exists. Only allowed if O_CREAT is set.
• **O_TRUNC**	Truncate the file to zero length before writing to it.

If successful, **open()** returns a file descriptor, a small integer used by other I/O functions to reference the file. Otherwise, errno is set and -1 is returned.

ERROR CODES

EACCES	Either no execute permission for path directory, no write permission for parent directory, or no read or write permission for specified file.
EEXIST	O_CREAT and O_EXCL are set and the specified file exists.
EFAULT	***path*** equals NULL.
EINVAL	The specified file name is invalid.
EISDIR	The specified file name is a directory.
ENAMETOOLONG	Length exceeds **PATH_MAX** and **_PATH_NO_TRUNC** is TRUE.
EMFILE	No file control block is free. FOPEN_MAX files are currently open.
ENOENT	Either parent directory not found or file does not exist and O_CREAT was not set.
ENOMEM	Unable to allocate memory to create file entry.

ENOSPC	No space is left on volume to create file.
ENOTDIR	A component of the path prefix is not a directory.
ENXIO	The volume specified by the path root name is not mounted.

EXAMPLE

```
fn = open("/flash/temp.txt", O_RDONLY);
if (fn == -1)
  perror("error opening the file!");
```

FAT • FFS • RFS • ZFS

opendir()

PROTOTYPE

```
#include <posix.h>

DIR *opendir(const char *dirname);
```

DESCRIPTION

opendir() is used to open a directory to access its file list using **readdir()**. List access is positioned at the first entry. ***dirname*** must be a valid directory path, absolute if starting with '/', otherwise relative to the current working directory.

If successful, **opendir()** returns a pointer to directory stream. Otherwise, errno is set and NULL is returned.

ERROR CODES

EACCES	No execute permission for directory in path prefix or no read permission for specified directory.
ENAMETOOLONG	**_PATH_NO_TRUNC** is TRUE and the length exceeds **PATH_MAX**.
ENOENT	The directory specified by ***dirname*** was not found.
ENOMEM	DIROPEN_MAX directories are currently open.
ENOTDIR	***dirname*** does not refer to a directory.
ENXIO	The volume specified by the path root name is not mounted.

EXAMPLE

```
/*-------------------------------------------------------------*/
/* Open directory associated with path1.                       */
/*-------------------------------------------------------------*/
curr_dir = opendir(path1);
if (curr_dir == NULL)
{
  perror("error in opening directory!");
  return –1;
}
```

FAT • FFS • RFS

perror()

PROTOTYPE

```
#include <stdio.h>

void perror(const char *s);
```

DESCRIPTION

perror() is used to output an error message corresponding to errno. If ***s*** is not NULL, the NULL terminated string it specifies is written to stderr followed by the two-character string ": ". Next, the error string corresponding to errno is written. Lastly, a terminating NL character is written.

EXAMPLE

```
/*-------------------------------------------------------------*/
/* Open directory associated with path1.                       */
/*-------------------------------------------------------------*/
curr_dir = opendir(path1);
if (curr_dir == NULL)
  perror("error in opening directory!");
```

FAT • FFS • RFS

printf()

PROTOTYPE

```
#include <stdio.h>

int printf(const char *format, …);
```

DESCRIPTION

printf() is used to write formatted text to the I/O stream stdout. ***format*** and any following arguments are interpreted according to Standard C's output format string conventions, which are documented in Appendix B.

If successful, **printf()** returns the number of characters written to stdout. Otherwise, errno is set and a negative value is returned.

ERROR CODES

EACCES	stdout is not open in append or write mode.
EBADF	stdout is invalid.
EIO	An output error occurred.
ENOSPC	Volume is full.

EXAMPLE

```
/*-----------------------------------------------------------------*/
/* Write simple message to output.                                 */
/*-----------------------------------------------------------------*/
if (printf("Hello World!\n") == -1)
{
  perror("error writing string to output");
  return -1;
}
```

FAT • FFS • RFS

putc()

PROTOTYPE

```
#include <stdio.h>

int putc(int ch, FILE *stream);
```

DESCRIPTION

putc() is used to write one character to the I/O stream identified by ***stream***. The character ***ch*** is converted to type “unsigned char” and written at the current file position, advancing the file position indicator, if defined. The function is equivalent to **fputc(ch, stream)**.

If successful, **putc()** returns the character written. Otherwise, errno is set and EOF is returned.

ERROR CODES

EACCES	File is not open in append or write mode.
EBADF	***stream*** is invalid.
EIO	An output error occurred.
ENOSPC	Volume is full.

EXAMPLE

```
stream_a = fopen("/flash/temp.txt", "r");

/*---------------------------------------------------------------*/
/* Output ‘H’ to stream_a.                                       */
/*---------------------------------------------------------------*/
if (putc(‘H’, stream_a) == EOF)
{
  perror("error writing to file!");
  return –1;
}
```

FAT • FFS • RFS

putchar()

PROTOTYPE

```
#include <stdio.h>

int putchar(int ch);
```

DESCRIPTION

putchar() is used to write one character to the I/O stream stdout. The character ***ch*** is converted to type “unsigned char” and written at the current file position, advancing the file position indicator, if defined. The function is equivalent to **fputc(ch, stdout)**.

If successful, **putchar()** returns the character written. Otherwise, errno is set and EOF is returned.

ERROR CODES

EACCES	stdout is not open in append or write mode.
EBADF	stdout is invalid.
EIO	An output error occurred.
ENOSPC	Volume is full.

EXAMPLE

```
/*------------------------------------------------------------*/
/* Echo stdin to stdout until ‘q’ is entered.                 */
/*------------------------------------------------------------*/
do
{
  c = getchar();
  if (putchar(c) == EOF)
    perror("error writing to output!");
} while (c != ‘q’);
```

FAT • FFS • RFS

puts()

PROTOTYPE

```
#include <stdio.h>

int puts(const char *string);
```

DESCRIPTION

puts() is used to write a string to the I/O stream stdout. ***string*** points to a NULL terminated string. All characters up to the terminating NULL are written, followed by a terminating NL character.

If successful, **puts()** returns a nonnegative value. Otherwise, errno is set and EOF is returned.

ERROR CODES

EACCES	stdout is not open in append or write mode.
EINVAL	***string*** equals NULL.
EBADF	stdout is invalid.
EIO	An output error occurred.
ENOSPC	Volume is full.

EXAMPLE

```
char s[13] = "Hello World!";

/*-----------------------------------------------------------------*/
/* Write s to stdout.                                              */
/*-----------------------------------------------------------------*/
if (puts(s) == EOF)
{
  perror("error writing to standard output!");
  return -1;
}
```

FAT • FFS • RFS • ZFS

read()

PROTOTYPE

```
#include <posix.h>

int read(int fid, void *buf, unsigned int nbytes);
```

DESCRIPTION

read() is used to read from the open file identified by descriptor ***fid***. ***buf*** points to a receive buffer. Characters are copied from the file to the buffer until ***nbytes*** characters are read or end-of-file is reached.

If any bytes are successfully read, **read()** returns number of bytes read. If no bytes were read, errno is set and -1 is returned.

ERROR CODES

EACCES	File is not open in read mode.
EBADF	***fid*** is invalid.
EFAULT	***buf*** equals NULL.
EIO	An input error occurred.

EXAMPLE

```
char string[12];
int fid;

/*------------------------------------------------------------------*/
/* Open the file in read-only mode.                                 */
/*------------------------------------------------------------------*/
fid = open("/flash/temp.txt", O_RDONLY);
  return -1;

/*------------------------------------------------------------------*/
/* Read 12 characters from the file into temporary buffer.          */
/*------------------------------------------------------------------*/
if (read(fid, string, 12) == EOF)
  return -1;
```

FAT • FFS • RFS • ZFS

readdir()

PROTOTYPE

```
#include <posix.h>

struct dirent *readdir(DIR *dirp);
```

DESCRIPTION

readdir() is used to read an entry in a directory's file list, advancing the position indicator. ***dirp*** must be a directory handle returned by **opendir()**. **readdir()** returns either a pointer to a dirent structure containing a file name or NULL if the end of the file list has been reached. The dirent structure is defined in "posix.h" and reproduced below:

```
struct dirent
{
  long d_ino;
  char d_name[FILENAME_MAX];
};
```

The dirent structure pointed to by the return value of **readdir()** may be overwritten by another **readdir()** call for the same directory.

If successful, **readdir()** returns pointer to the **dirent** structure. Otherwise, errno is set and NULL is returned.

ERROR CODES

EBADF	The directory handle ***dirp*** is invalid.

EXAMPLE

```
/*-------------------------------------------------------------*/
/* Print the name of every file in the directory.              */
/*-------------------------------------------------------------*/
while (entry = readdir(curr_dir))
  printf("%.*s\n", FILENAME_MAX, entry->d_name);
```

FAT • FFS • RFS

remove()

PROTOTYPE

```
#include <stdio.h>

int remove(const char *path);
```

DESCRIPTION

remove() is used to remove a file from a directory and decrement its link count. If the link count goes to zero, the file is deleted. ***path*** specifies the pathname for the file to be removed. Pathnames are absolute if they start with '/', otherwise they are relative to the current working directory.

If successful, **remove()** returns 0. Otherwise, errno is set and a nonzero value is returned.

ERROR CODES

EACCES	No execute permission for directory in path prefix or no write permission for specified directory.
EBUSY	The file specified by ***path*** is currently open.
EEXIST	***path*** specifies a directory that is not empty.
EFAULT	***path*** equals NULL.
ENAMETOOLONG	Length exceeds **PATH_MAX** and **_PATH_NO_TRUNC** is TRUE.
ENOENT	The file specified by ***path*** was not found.
ENOTDIR	A component of the path prefix is not a directory.
ENXIO	The volume specified by the path root name is not mounted.
EPERM	User ID determined by **FsGetId()** does not match file's owner.
ENXIO	The volume specified by the path root name is not mounted.

EXAMPLE

```
/*---------------------------------------------------------------*/
/* Remove the specified file.                                    */
/*---------------------------------------------------------------*/
if (remove(path1))
  perror("error trying to remove file!");
```

FAT • FFS • RFS

rename()

PROTOTYPE

```
#include <stdio.h>

int rename(const char *old, const char *new);
```

DESCRIPTION

rename() is used to change the name of a directory or file. If successful, the name is changed from pathname ***old*** to pathname ***new***. Pathnames are absolute if they start with '/', otherwise they are relative to the current working directory. The new file or directory must be on the same file system as the old file or directory. It is an error if the file or directory specified by **new** already exists.

If successful, **rename()** returns 0. Otherwise, errno is set and -1 is returned.

ERROR CODES

EACCES	No execute permission for path directory or no write permission for parent directory.
EEXIST	***new*** specifies an existing file.
EFAULT	Either ***old*** or ***new*** equals NULL.
EINVAL	Either ***old*** or ***new*** are invalid path names or ***old*** is a parent directory of ***new***.
ENAMETOOLONG	Length exceeds **PATH_MAX** and **_PATH_NO_TRUNC** is TRUE.
ENOENT	The file or directory specified by ***old*** was not found.
ENOSPC	No space is left on volume to rename a file or directory.
ENOTDIR	A component of one of the path prefixes is not a directory.
EPERM	User ID determined by **FsGetId()** does not match file's owner.
EXDEV	***new*** is not on the same file system as ***old***.
ENXIO	The volume specified by a path root name is not mounted.

EXAMPLE

```
/*-------------------------------------------------------------*/
/* Rename the file or directory.                               */
/*-------------------------------------------------------------*/
if (rename(path1, path2))
  perror("error trying to rename file!");
```

FAT • FFS • RFS • ZFS

rewind()

PROTOTYPE

```
#include <stdio.h>

void rewind(FILE *stream);
```

DESCRIPTION

rewind() is used to clear the error indicator and reset the file position for the I/O stream identified by ***stream***. A call to **rewind(*stream*)** is like a call to **fseek(*stream, 0, SEEK_SET*)** except that the error indicator is also cleared.

On error, errno is set to a positive value.

ERROR CODES

EBADF	***stream*** is invalid.

EXAMPLE

```
/*-----------------------------------------------------------------*/
/* Rewind stream_a in order to read from the beginning.           */
/*-----------------------------------------------------------------*/
errno = 0;
rewind(stream_a);
if (errno)
  perror("error trying to rewind file!");
```

FAT • FFS • RFS • ZFS

rewinddir()

PROTOTYPE

```
#include <posix.h>

void rewinddir(DIR *dirp);
```

DESCRIPTION

rewinddir() is used to reset a directory's **readdir()** pointer to the first file in the directory. ***dirp*** must be a pointer returned by **opendir()** that has not been closed by **closedir()**.

If an error occurs, errno is set to the appropriate error code.

ERROR CODES

EBADF	***dirp*** is invalid.

EXAMPLE

```
/*-------------------------------------------------------------*/
/* Reset the directory position indicator.                     */
/*-------------------------------------------------------------*/
rewinddir(curr_dir);
```

FAT • FFS • RFS

rmdir()

PROTOTYPE

```
#include <posix.h>

int rmdir(const char *path);
```

DESCRIPTION

rmdir() is used to remove an empty directory. ***path*** must be a valid directory pathname, absolute if starting with '/', otherwise relative to the current working directory.

If successful, **rmdir()** returns 0. Otherwise, errno is set and -1 is returned.

ERROR CODES

EACCES	No execute permission for path directory or no write permission for parent directory.
EBUSY	The directory specified by ***path*** is either open or the current working directory of another task.
EEXIST	The directory specified by ***path*** is not empty.
EFAULT	***path*** equals NULL.
ENAMETOOLONG	Length exceeds **PATH_MAX** and **_PATH_NO_TRUNC** is TRUE.
ENOENT	The directory specified by ***path*** was not found.
ENOTDIR	A component of the path prefix is not a directory.
ENOTEMPTY	The specified directory is either not empty or is the current working directory.
ENXIO	The volume specified by the path root name is not mounted.
EPERM	User ID determined by **FsGetId()** does not match file's owner.
ENXIO	The volume specified by the path root name is not mounted.

EXAMPLE

```
/*---------------------------------------------------------------*/
/* Remove the named directory.                                   */
/*---------------------------------------------------------------*/
if (rmdir(path1))
  perror("error trying to remove directory!");
```

FAT • FFS • RFS • ZFS

scanf()

PROTOTYPE

```
#include <stdio.h>

int scanf(const char *format, …);
```

DESCRIPTION

scanf() is used to read formatted input from the I/O stream stdin. ***format*** is interpreted according to Standard C's input format string conventions, which are documented in Appendix B. Any additional arguments are variable addresses into which values are assigned.

Under control of the format string, characters are read from the input stream, converted, and written to the provided variable addresses. Processing continues until the terminating NULL in the format string is reached, end-of-file is reached, or a conversion error occurs.

If no input assignments are made before either end-of-file is reached or a conversion error occurs, errno is set and **scanf()** returns EOF. Otherwise, **scanf()** returns the number of input assignments.

ERROR CODES

EACCES	stdin is not open in read mode.
EBADF	stdin is invalid.
EIO	An input error occurred.
EISDIR	stdin refers to a directory.

EXAMPLE

```
int i;

/*----------------------------------------------------------------*/
/* Read an integer from stdin.                                    */
/*----------------------------------------------------------------*/
if (scanf("%d", &i) == EOF)
{
  perror("error reading from file");
  return -1;
}
```

FAT • FFS • RFS • ZFS

setbuf()

PROTOTYPE

```
#include <stdio.h>

void setbuf(FILE *stream, char *buf);
```

DESCRIPTION

setbuf() is used in some environments to control how a stream will be buffered. A call to “**set-buf(stream, buf)**” is equivalent to calling **”setvbuf(file, buf, _IOFBF, BUFSIZ);**”

The TargetFFS buffer size is set at compile time. **setbuf()** has no effect.

FAT • FFS • RFS • ZFS

setvbuf()

PROTOTYPE

```
#include <stdio.h>

int setvbuf(FILE *stream, char *buf, int mode, size_t size);
```

DESCRIPTION

setvbuf() is used in some environments to control how a stream will be buffered. ***buf*** points to a user-allocated buffer. ***size*** is the size of this buffer. ***mode*** is one of:

_IOFBF - for fully buffered I/O
_IOLBF - for line buffered I/O
_IONBF - for unbuffered I/O

The TargetFFS buffer size is set at compile time. **setvbuf()** has no effect.

If successful, **setvbuf()** returns 0. Otherwise, errno is set and a non-zero value is returned.

ERROR CODES

EBADF	***stream*** is invalid.
EINVAL	***mode*** is invalid or the request cannot be honored.

FAT • FFS • RFS • ZFS

stat()

PROTOTYPE

```
#include <posix.h>

int stat(const char *path, struct stat *buf);
```

DESCRIPTION

stat() is used to obtain information about a file or directory. ***path*** must be a valid path name, absolute if starting with '/', otherwise relative to the current working directory. The status information is written to the struct stat structure pointed to by ***buf***. The stat structure is defined in "posix.h" and reproduced below:

```
struct stat
{
  mode_t  st_mode;   /* file mode */
  ino_t   st_ino;    /* file serial number */
  dev_t   st_dev;    /* ID of device containing this file */
  nlink_t st_nlink;  /* number of links */
  uid_t   st_uid;    /* user ID of file's owner */
  gid_t   st_gid;    /* group ID of file's owner */
  off_t   st_size;   /* the file size in bytes */
  time_t  st_atime;  /* time of last access */
  time_t  st_mtime;  /* time of last data modification */
  time_t  st_ctime;  /* time of last status change */
};
```

The macros S_ISDIR() amd S_ISREG() can be applied to the st_mode field to determine whether ***path*** specifies a directory or a regular file.

If successful, **stat()** returns 0. Otherwise, errno is set and -1 is returned.

ERROR CODES

EACCES	No execute permission for directory in path prefix.
EFAULT	Either ***buf*** or ***path*** equals NULL.
ENAMETOOLONG	Length exceeds **PATH_MAX** and **_PATH_NO_TRUNC** is TRUE.
ENOENT	The file or directory specified by ***path*** was not found.
ENOTDIR	A component of the path prefix is not a directory.
ENXIO	The volume specified by the path root name is not mounted.

EXAMPLE

```
struct stat buf;

if (stat(path1, &buf))
  perror("error trying to read stats for file!");
else
{
  printf("File access time:   %s", ctime(&buf.st_atime));
  printf("File modified time: %s", ctime(&buf.st_mtime));
  printf("File mode:          %d\n", buf.st_mode);
  printf("File serial number: %d\n", buf.st_ino);
  printf("File num links:     %d\n", buf.st_nlink);
  printf("File user ID:       %u\n", buf.st_uid);
  printf("File group ID:      %u\n", buf.st_gid);
  printf("File size (byte):   %u\n", buf.st_size);
}
```

FAT • FFS • RFS

sync()

PROTOTYPE

```
#include <posix.h>

int sync(int fid);
```

DESCRIPTION

sync() is used to flush buffered data, writing it to the underlying storage media. If ***fid*** is -1, the buffered data for every open file is flushed. Otherwise, just data for the specified file is flushed.

If successful, **sync()** returns 0. Otherwise, errno is set and EOF is returned.

ERROR CODES

EBADF	The file descriptor ***fid*** is invalid.

EXAMPLE

```
/*-----------------------------------------------------------------*/
/* Flush file data to the flash media.                             */
/*-----------------------------------------------------------------*/
if (sync(fid))
  return –1;
```

RFS

tmpfile()

PROTOTYPE

```
#include <stdio.h>

FILE *tmpfile(void);
```

DESCRIPTION

tmpfile() is used to create a temporary file, opened with "wb+" mode, that will be automatically removed when either the file is closed or the program terminates. **tmpfile()** requires the TargetOS RAM file system.

If successful, **tmpfile()** returns a pointer to the temporary file. Otherwise, errno is set and NULL is returned.

ERROR CODES

EACCES	No write permission for the RAM root directory.
EINVAL	The TargetOS RAM file system is not mounted.
EMFILE	No file control block is free. FOPEN_MAX files are currently open.
ENOSPC	No space is left on volume to create file.

EXAMPLE

```
FILE *stream_tmp;

/*-------------------------------------------------------------------*/
/* Open a temporary file to store the info in.                       */
/*-------------------------------------------------------------------*/
stream_tmp = tmpfile();
if (stream_tmp == NULL)
{
  perror("error opening file!");
  return -1;
}
```

FFS • RFS

tmpnam()

PROTOTYPE

```
#include <stdio.h>

char *tmpnam(char *s);
```

DESCRIPTION

tmpnam() is used to create a unique filename. If ***s*** is not NULL, a unique filename of up to L_tmpnam number of characters will be stored into it. Also, the name is stored in a static array whose address is returned by **tmpnam()**. Subsequent calls to **tmpnam()** may alter the contents of the array.

tmpnam() returns unique names for the first TMP_MAX calls, after that uniqueness is not guaranteed.

EXAMPLE

```
char temp_name[L_tmpnam];

/*-----------------------------------------------------------------*/
/* Get a unique name for a file.                                   */
/*-----------------------------------------------------------------*/
tmpnam(temp_name);
```

FAT • FFS • RFS

truncate()

PROTOTYPE

```
#include <posix.h>

int truncate(const char *path, off_t length);
```

DESCRIPTION

truncate() is used to set the length of the specified regular file to ***length*** characters. ***path*** must be a valid path name, absolute if starting with '/', otherwise relative to the current working directory. If ***length*** is greater than the file's current length, the file is extended by appending '\0' characters. Otherwise, the file is truncated to ***length*** characters. The file's position indicator is set to the end of the file.

If successful, **truncate()** returns 0. Otherwise, errno is set and -1 is returned.

ERROR CODES

EACCES	Either no execute permission for a path directory or no write permission for the specified file.
EFAULT	***path*** equals NULL.
EINVAL	***length*** is negative or the file name is invalid.
EIO	An output error occurred.
EISDIR	The specified file name is a directory.
ENAMETOOLONG	Length exceeds **PATH_MAX** and **_PATH_NO_TRUNC** is TRUE.
ENOENT	Either parent directory not found or file does not exist.
ENOSPC	No space is left on volume to write file data.
ENOTDIR	A component of the path prefix is not a directory.
ENXIO	The volume specified by the path root name is not mounted.

EXAMPLE

```
/*-------------------------------------------------------------------*/
/* Clip the named file to 100 bytes.                                 */
/*-------------------------------------------------------------------*/
if (truncate("/flash/rhello.c", 100))
  perror("error trying to truncate file!");
```

FAT • FFS • RFS • ZFS

ungetc()

PROTOTYPE

```
#include <stdio.h>

int ungetc(int ch, FILE *stream);
```

DESCRIPTION

ungetc() is used to push back a character to a file, so that it is read by the next input operation. ***ch*** is converted to an unsigned character and put back onto ***stream***. If **ungetc()** is called twice without an intervening read operation, the second **ungetc()** will fail. Attempts to push back EOF have no effect.

ungetc() returns ***ch*** if no other character was pushed back since the last read, EOF otherwise.

ERROR CODES

EBADF	***stream*** is invalid.

EXAMPLE

```
stream_a = fopen("/flash/temp.txt", "r");

/*-------------------------------------------------------------*/
/* Output the first letter in stream_a 12 times.               */
/*-------------------------------------------------------------*/
for (count = 0; count < 12; ++count)
{
  c = getc(stream_a);
  printf("%c", c);
  ungetc(c, stream_a);
}
```

FAT • FFS • RFS

unlink()

PROTOTYPE

```
#include <posix.h>

int unlink(const char *path);
```

DESCRIPTION

unlink() is used to remove a file from a directory and decrement its link count. If the link count goes to zero, the file is deleted. ***path*** specifies the pathname for the file to be removed. Pathnames are absolute if they start with '/', otherwise they are relative to the current working directory.

If successful, **unlink()** returns 0. Otherwise, errno is set and -1 is returned.

ERROR CODES

EACCES	No execute permission for path directory or no write permission for specified directory.
EBUSY	The file specified by ***path*** is currently open.
EEXIST	***path*** specifies a directory that is not empty.
EFAULT	***path*** equals NULL.
ENAMETOOLONG	Length exceeds **PATH_MAX** and **_PATH_NO_TRUNC** is TRUE.
ENOENT	The file specified by ***path*** was not found.
ENOTDIR	A component of the path prefix is not a directory.
ENXIO	The volume specified by the path root name is not mounted.
EPERM	User ID determined by **FsGetId()** does not match file's owner.
ENXIO	The volume specified by the path root name is not mounted.

EXAMPLE

```
/*-------------------------------------------------------------*/
/* Unlink the file path1.                                      */
/*-------------------------------------------------------------*/
if (unlink(path1))
  perror("error trying to unlink file!");
```

FAT • FFS • RFS • ZFS

unmount()

PROTOTYPE

```
#include <posix.h>

int unmount(char *path);
```

DESCRIPTION

unmount() is used to unmount the volume specified by **path**. After a volume has been unmounted, its files are inaccessible to applications. Any open files on the volume are closed after their contents are flushed to the backing store. **path** must match the root name of a mounted volume.

When a volume is unmounted, any resources allocated to support application use of the volume's files are freed. An unmounted volume uses a minimum of system resources. A volume may be mounted and unmounted as necessary to manage use of system resources.

unmount() returns zero if successful, -1 on error.

ERROR CODES

EFAULT	***path*** equals NULL.
ENAMETOOLONG	Length exceeds **PATH_MAX** and **_PATH_NO_TRUNC** is TRUE.
ENOENT	No volume matching the specified name has been mounted.

EXAMPLE

```
/*-------------------------------------------------------------------*/
/* Unmount the flash file system and check for success.             */
/*-------------------------------------------------------------------*/
if (unmount("flash"))
  printf("unmount(\”flash\”) failed!\n");
```

FAT • FFS • RFS

utime()

PROTOTYPE

```
#include <posix.h>

int utime(const char *path, const struct utimbuf *times);
```

DESCRIPTION

utime() is used to set the access and modification times for a file. ***path*** specifies the file's path-name. Pathnames are absolute if they start with '/', otherwise they are relative to the current working directory.

If ***times*** is NULL, the file's access and modification times are set to the current time. Otherwise, the values stored in the utimbuf structure pointed at by ***times*** are used. The utimbuf structure is defined in "posix.h" and reproduced below:

```
struct utimbuf
{
  time_t actime;   /* access time */
  time_t modtime;  /* modification time */
};
```

If successful, **utime()** returns 0. Otherwise, errno is set and -1 is returned.

ERROR CODES

EACCES	No execute permission for directory in path prefix.
EFAULT	***path*** is not a valid pointer.
ENAMETOOLONG	Length exceeds **PATH_MAX** and **_PATH_NO_TRUNC** is TRUE.
ENOENT	The file specified by ***path*** was not found.
ENOTDIR	A component of the path prefix is not a directory.
EPERM	User ID determined by **FsGetId()** does not match file's owner.
ENXIO	The volume specified by the path root name is not mounted.

EXAMPLE

```
/*-----------------------------------------------------------------*/
/* Set the access and modification times to the current time.      */
/*-----------------------------------------------------------------*/
utime(upgrade, NULL);
```

FFS

vclean()

PROTOTYPE

```
#include <posix.h>

int vclean(const char *path);
```

DESCRIPTION

vclean() is used to perform recycles, which convert dirty sectors to free sectors, in advance. It can be used to increase the size of the free sector pool. ***path*** is the root name of the affected volume. Normally the file system performs "just in time" recycling by checking before each free sector allocation to see if the free sector count has reached a lower threshold.

When used with EXTRA_FREE, **vclean()** can ensure a pool of free sectors is always available for user data, even when a volume is full or nearly full. When used with **vstat()**, **vclean()** provides support for atomic file updates. These applications are discussed in Chapter 2.

vclean() returns -1 if an error has occurred, 0 if no additional dirty sectors can be freed, and 1 if there are additional dirty sectors to free. To avoid 'hogging' access to the file system, each call to **vclean()** performs at most a single recycle.

ERROR CODES

EFAULT	***path*** equals NULL.
ENAMETOOLONG	Length exceeds **PATH_MAX** and **_PATH_NO_TRUNC** is TRUE.
ENOENT	No volume matching the specified name has been mounted.

EXAMPLE

```
/***********************************************************************/
/*     cleaner: Reclaim dirty sectors in 'background'.                 */
/*                                                                     */
/***********************************************************************/
static void cleaner(void)
{
  for (;;)
  {
    while (vclean("flash") > 0) ;
    taskSleep(2 * OsTicksPerSec);
  }
}
```

FAT • FFS • RFS

vfprintf()

PROTOTYPE

```
#include <stdio.h>

int vfprintf(FILE *stream, const char *format, va_list ap);
```

DESCRIPTION

vfprintf() is used to print formatted text to the I/O stream identified by ***stream***. ***format*** and the context information provided by ***ap*** are interpreted according to Standard C's output format string conventions, as documented in Appendix B.

If successful, **vfprintf()** returns the number of characters written to ***stream***. Otherwise, errno is set and -1 is returned.

ERROR CODES

EACCES	File is not open in append or write mode.
EBADF	***stream*** is invalid.
EIO	An output error occurred.
ENOSPC	Volume is full.

EXAMPLE

```
/*-------------------------------------------------------------------*/
/* Simple function that is the equivalent of printf with error output */
/*-------------------------------------------------------------------*/
int print(const char *format, ...)
{
  va_list ap;
  int r_value;

  va_start(ap, format);
  if ((r_value = vfprintf(stdout, format, ap)) < 0)
    perror("error printing to stdout!");
  va_end(ap);
}
```

FAT • FFS • RFS

vprintf()

PROTOTYPE

```
#include <stdio.h>

int vprintf(const char *format, va_list arg);
```

DESCRIPTION

vprintf() is used to print formatted text to the I/O stream stdout. ***format*** and the context information provided by ***ap*** are interpreted according to Standard C's output format string conventions, as documented in Appendix B.

If successful, **vprintf()** returns the number of characters written to stdout. Otherwise, errno is set and -1 is returned.

ERROR CODES

EACCES	stdout is not open in append or write mode.
EBADF	stdout is invalid.
EIO	An output error occurred.
ENOSPC	Volume is full.

EXAMPLE

```
/*-------------------------------------------------------------------*/
/* Simple function that is the equivalent of printf with error output */
/*-------------------------------------------------------------------*/
int print(const char *format, ...)
{
  va_list ap;
  int r_value;

  va_start(ap, format);
  if ((r_value = vprintf(format, ap)) < 0)
    perror("error printing to stdout!");
  va_end(ap);
}
```

FAT • FFS • RFS • ZFS

vstat()

PROTOTYPE

```
#include <posix.h>

int vstat(const char *path, union vstat *buf);
```

DESCRIPTION

vstat() is used to obtain information about the volume specified by **path**. ***buf*** points to a union that contains a unique structure definition for each TargetOS file system. See "Volume Information" in Chapter 2 for details on the structure definition used with this file system.

vstat() returns zero if successful, -1 on error.

ERROR CODES

EFAULT	Either ***path*** or ***buf*** equals NULL.
ENAMETOOLONG	Length exceeds **PATH_MAX** and **_PATH_NO_TRUNC** is TRUE.
ENOENT	No volume matching the specified name has been mounted.

EXAMPLE

```
union vstat;

if (vstat("flash", &vstat))
  printf("vstat() on \”flash\” failed!\n");
else
  printf(“wear_count = %u\n”, vstat.ffs.wear_count);
```

FAT • FFS • RFS

write()

PROTOTYPE

```
#include <posix.h>

int write(int fid, const void *buf, unsigned int nbytes);
```

DESCRIPTION

write() is used to write to the open file identified by the descriptor ***fid***. ***buf*** points to a write buffer. Characters are copied from the buffer to the file until ***nbytes*** characters are written or an error occurs.

If any bytes are successfully written, **write()** returns number of bytes written. If no bytes were written, errno is set and -1 is returned.

ERROR CODES

EACCES	File is not open in append or write mode.
EBADF	***fid*** is invalid.
EFAULT	***buf*** is not a valid pointer.
EIO	An output error occurred.
ENOSPC	No space is left on volume to write file data.

EXAMPLE

```
int fid;
char str1[14] = "Hello World!\n";

/*-------------------------------------------------------------------*/
/* Open a text file in write only mode.                              */
/*-------------------------------------------------------------------*/
fid = open("/ram_fs/a/b/file1.txt", O_WRONLY);

/*-------------------------------------------------------------------*/
/* Write the contents of str1 to the file.                           */
/*-------------------------------------------------------------------*/
if (write(fid, str1, 13) != 13)
{
  perror("error writing to file!");
  return -1;
}
```

Flash Driver Interface 4

Introduction

This chapter describes TargetFFS-NAND's driver interface. The interface is implemented using a combination of TargetFFS-NAND routines and driver callback routines. Four TargetFFS-NAND routines are called by flash file system drivers. Six driver callback routines are called by TargetFFS-NAND. The callback routines are registered with TargetFFS-NAND when the driver calls **FsNandAddVol()**.

The TargetFFS-NAND routines used by flash file system drivers are:

- DecodeEcc()
- EncodeEcc()
- FlashDelVol()
- FsNandAddVol()

The flash driver callback functions used by TargetFFS-NAND are:

- erase_block()
- page_erased()
- read_page()
- read_type()
- write_page()
- write_type()

The documentation for each routine includes its Standard C prototype and a description of its operation and use. Examples are given in the next chapter, which contains a complete NAND flash driver.

DecodeEcc()

PROTOTYPE

```
#include <fsprivate.h>

int DecodeEcc(ui32 *data, const ui8 *ecc);
```

DESCRIPTION

DecodeEcc() is a TargetFFS-NAND function used by drivers in **read_page()**, to check a NAND page for errors and correct them, if possible. ***data*** points to a 4-byte aligned array of 512 bytes of page data. ***ecc*** is a 10-byte array of Error Correction Code (ECC) values read from the page's extra bytes. If the NAND page size is bigger than 512 bytes, **DecodeEcc()** can be called multiple times, once for each 512 byte segment and 10-byte ECC encoding.

To use **DecodeEcc()**, ***ecc*** must have been calculated by **EncodeEcc()**. The TargetFFS-NAND ECC is fast and efficient, detecting up to eight bit errors and correcting up to four bit errors, but if a hardware calculate ECC is used in place of **EncodeEcc()**, a corresponding custom decoding routine should be used in place of **DecodeEcc()**.

DecodeEcc() returns 0 if the page was left with no errors and -1 if the page had errors it was unable to correct.

EXAMPLE

```
    /*------------------------------------------------------------------*/
    /* Perform ECC decoding, return error if uncorrectable error.        */
    /*------------------------------------------------------------------*/
    return DecodeEcc(buffer, ecc);
  }
```

EncodeEcc()

PROTOTYPE

```
#include <fsprivate.h>

void EncodeEcc(const ui32 *data, ui8 *ecc);
```

DESCRIPTION

EncodeEcc() is a TargetFFS-NAND function used by drivers in **write_page()**, to generate an Error Correction Code (ECC) that is written into the NAND page's extra bytes. ***data*** points to a 4-byte aligned array of 512 bytes of page data. ***ecc*** points to a 10-byte array that the ECC is written into. **write_page()** writes this array into the NAND page's extra bytes. If the NAND page size is bigger than 512 bytes, **EncodeEcc()** can be called multiple times, generating 10-bytes of ECC encoding for each 512 byte segment.

The TargetFFS-NAND ECC is fast and efficient, detecting up to eight bit errors and correcting up to four bit errors, but if a hardware calculated ECC is used instead, the **write_page()** call to **EncodeEcc()** would be replaced by accesses to special hardware registers. A custom decoding routine, not **DecodeEcc()**, would then be used in **read_page()**.

EXAMPLE

```
/***********************************************************************/
/*  write_page: ECC encode and write one NAND flash page              */
/*                                                                     */
/*      Inputs: buffer = pointer to buffer containing data to write    */
/*              type = page type flag                                  */
/*              addr = the page address                                */
/*              wear = wear count of this page's block                 */
/*              vol = driver's volume pointer                          */
/*                                                                     */
/*     Returns: 0 for success, -1 for chip error, 1 for verify error   */
/*                                                                     */
/***********************************************************************/
static int write_page(void *buffer, ui32 addr, ui32 type, ui32 wear,
                      void *vol)
{
  int n;
  ui8 *dst, ecc[10], status;

  /*-----------------------------------------------------------------*/
  /* Compute page's ECC encoding.                                    */
  /*-----------------------------------------------------------------*/
  EncodeEcc(buffer, ecc);
```

erase_block()

PROTOTYPE

```
#include <fsprivate.h>

int erase_block(ui32 addr, void *vol);
```

DESCRIPTION

erase_block() is a driver callback function used by TargetFFS-NAND to erase the block of flash memory at the base address specified by ***addr***. A block is the minimum “chunk” of flash memory that can be erased. Its value depends on the flash device used and how it is accessed on the target board.

vol can be used to support multiple volumes. The value assigned to the "vol" field in the FfsVol structure that is passed to TargetFFS-NAND by **FsNandAddVol()** is passed unmodified as the last parameter in the flash driver callback functions. This field may be initialized to point to a structure containing volume-specific information, allowing one set of driver callback functions to support multiple volumes.

erase_block() should not return until erasure is complete. It can poll a flash device's “ready” bit or, on systems that support this, block until an interrupt from the flash device signals that the operation is complete.

erase_block() returns 0 for success or -1 if an error occurs. The block is marked "bad" by TargetFFS-NAND if **erase_block()** returns -1. Once a block is marked bad, it is no longer erased or written.

FlashDelVol()

PROTOTYPE

```
#include <fsprivate.h>

int FlashDelVol(const char *name);
```

DESCRIPTION

FlashDelVol() is a TargetFFS-NAND function used by drivers or applications to uninstall a flash volume. If the volume is mounted when **FlashDelVol()** is called, it is first unmounted. Any open files on the volume are closed. The volume can not be mounted or formatted until after a subsequent call to **FsNandAddVol()**.

FlashDelVol() returns 0 for success or -1 if an error occurs.

ERROR CODES

ENOENT	No currently installed volume matches the specified name.

EXAMPLE

```
/*-------------------------------------------------------------------*/
/* If unable to remove volume, return error.                         */
/*-------------------------------------------------------------------*/
if (FlashDelVol("flash2"))
  return –1;
```

FsNandAddVol()

PROTOTYPE

```
#include <fsprivate.h>

int FsNandAddVol(FfsVol *vol);
```

DESCRIPTION

FsNandAddVol() is a TargetFFS-NAND function used by drivers to register a flash volume with TargetFFS-NAND. A volume cannot be mounted or formatted until after **FsNandAddVol()** succeeds. ***vol*** points to a FfsVol structure which defines the volume's interface to TargetFFS-NAND. This structure is defined in "fsprivate.h" and reproduced below:

```
typedef struct
{
  char *name;         /* name of this volume */
  ui32 block_size;    /* minimum erasable block size in bytes */
  ui32 page_size;     /* page size in bytes */
  ui32 num_blocks;    /* number of blocks in volume */
  ui32 mem_base;      /* volume base address */
  void *vol;          /* driver's volume pointer */
  ui32 max_bad_blocks;
  FlashDriver driver;
} FfsVol;
```

All the fields in this structure must be initialized before calling **FsNandAddVol()**. The various fields are described below:

name is the volume name. It becomes the root of the path name for all files and directories created on the volume. It cannot match the name of any previously registered volume.

block_size is the number of bytes in the smallest erasable "chunk" of flash memory. Its value depends on the flash device used and how it is accessed on the target board.

page_size is the number of bytes of user data in the NAND device's page. Usually 512 bytes, its value depends on the flash device used.

num_blocks is the number of erasable blocks assigned to the volume. TargetFFS-NAND needs a minimum of four erasable blocks to create a flash volume. For best operation, at least eight or more blocks should be allocated. Normally this field would be set to the total number of erasable blocks in the device.

mem_base is the base address in the NAND address space of the volume's first erasable block. Normally, this would be zero. However, it is possible to reserve some blocks in a device by setting this field to the offset of the first block to be used by TargetFFS-NAND.

vol allows a single driver to support multiple volumes. Any value assigned to this field is passed unmodified back to the driver in every call made by TargetFFS-NAND, as the last parameter in the driver routine. If multiple volumes are implemented with similar hardware, the values unique to each volume, such as its base address in the processor address space, may be placed in a structure. The **FsNandAddVol()** call for each volume would point ***vol*** to the appropriate structure, allowing the driver routines to use their ***vol*** parameter to address the appropriate volume.

max_bad_blocks is the maximum number of bad blocks specified to exist in the device over the course of its guaranteed lifetime. Bad blocks can exist in new devices and also appear randomly during device operation. This value should be determined from the device's data sheet.

> NOTE: Once a volume has been formatted by TargetFFS-NAND, the settings for ***block_size***, ***page_size***, ***num_blocks***, and ***max_bad_blocks*** must remain unchanged in subsequent calls to **FsNandAddVol()**.

driver is a union whose "nand" member has type FsNandDriver. FsNandDriver is a structure consisting of pointers to the six callback functions provided by the driver. The structure is defined in “fsprivate.h” and reproduced below. Its six fields should be assigned the addresses of the six driver callback functions defined in this chapter.

```
typedef struct
{
  int  (*write_page)(void *buf, ui32 addr, ui32 type, ui32 wear, void *vol);
  int  (*write_type)(ui32 addr, ui32 type, void *vol);
  int  (*read_page)(void *buf, ui32 addr, ui32 wear, void *vol);
  ui32 (*read_type)(ui32 addr, void *vol);
  int  (*page_erased)(ui32 addr, void *vol);
  int  (*erase_block)(ui32 addr, void *vol);
} FsNandDriver;
```

FsNandAddVol() returns 0 for success or -1 if an error occurs.

ERROR CODES

EEXIST	An existing volume has the same name.
EINVAL	The volume name is invalid or ***num_blocks*** specifies less than three erasable blocks.
ENOMEM	The maximum number of volumes has already been added. See “Configuration” in Chapter 1.

EXAMPLE

```
/*-------------------------------------------------------------------*/
/* Register flash volume with TargetFFS-NAND.                        */
/*-------------------------------------------------------------------*/
vol.name = "flash";
vol.block_size = BLOCK_SIZE;
vol.page_size = 512;
vol.num_blocks = NUM_BLOCKS;
vol.mem_base = 0;
vol.vol = 0;
vol.max_bad_blocks = 10;   /* from data sheet */
vol.driver.nand.write_page = write_page;
vol.driver.nand.write_type = write_type;
vol.driver.nand.read_page = read_page;
vol.driver.nand.read_type = read_type;
vol.driver.nand.page_erased = page_erased;
vol.driver.nand.erase_block = erase_block;
if (FsNandAddVol(&vol))
  perror("FsNandAddVol() failed");
```

page_erased()

PROTOTYPE

```
#include <fsprivate.h>

int page_erased(ui32 addr, void *vol);
```

DESCRIPTION

page_erased() is a driver callback function used by TargetFFS-NAND to determine if the flash page at the address specified by ***addr*** is completely erased: both data bytes and spare bytes. It is an optimization in that the driver can usually be written to tell if a given page is erased faster than if TargetFFS-NAND read the page and then checked for all 0xFF's.

vol can be used to support multiple volumes. The value assigned to the "vol" field in the FfsVol structure that is passed to TargetFFS-NAND by **FsNandAddVol()** is passed unmodified as the last parameter in the flash driver callback functions. This field may be initialized to point to a structure containing volume-specific information, allowing one set of driver callback functions to support multiple volumes.

page_erased() returns TRUE if the page is completely erased, FALSE if any bit is 0.

read_page()

PROTOTYPE

```
#include <fsprivate.h>

int read_page(void *buf, ui32 addr, ui32 wear, void *vol);
```

DESCRIPTION

read_page() is a driver callback function used by TargetFFS-NAND to read the page of flash memory at address ***addr***. The size of a page is set by the "page_size" value in the FfsVol structure passed by **FsNandAddVol()**. ***buf*** points to the buffer that is to receive the page of data. This buffer is guaranteed to be aligned on a four-byte boundary.

wear is the wear count for the block containing the addressed page. For devices that guarantee zero bit errors below a specified wear value, ***wear*** may be used to omit ECC decoding, for higher performance. Otherwise, the data read from NAND memory must be error corrected before it is passed to TargetFFS-NAND.

Error correction is done using an error correction code (ECC) read from the page's extra bytes. The ECC is written to the extra bytes by **write_page()**. TargetFFS-NAND supplies two fast ECC routines, **EncodeEcc()** for use in **write_page()** and **DecodeEcc()** for use in **read_page()**. Alternately, if available, hardware ECC support may be used.

vol can be used to support multiple volumes. The value assigned to the "vol" field in the FfsVol structure that is passed to TargetFFS-NAND by **FsNandAddVol()** is passed unmodified as the last parameter in the flash driver callback functions. This field may be initialized to point to a structure containing volume-specific information, allowing one set of driver callback functions to support multiple volumes.

read_page() returns 0 for success and -1 if the data has an uncorrectable error.

read_type()

PROTOTYPE

```
#include <fsprivate.h>

ui32 read_type(ui32 addr, void *vol);
```

DESCRIPTION

read_type() is a driver callback function used by TargetFFS-NAND to read the type flag for the flash memory page at address ***addr***. The type flag is written by **write_page()** and may be updated by **write_type()**. The type flag is stored in the page's extra bytes and is used by TargetFFS-NAND to mark the page as containing either user data or TargetFFS-NAND control data. No ECC encoding or decoding is done to the type flag.

vol can be used to support multiple volumes. The value assigned to the "vol" field in the FfsVol structure that is passed to TargetFFS-NAND by **FsNandAddVol()** is passed unmodified as the last parameter in the flash driver callback functions. This field may be initialized to point to a structure containing volume-specific information, allowing one set of driver callback functions to support multiple volumes.

read_type() returns the four-byte type flag read from the flash page's extra bytes.

write_page()

PROTOTYPE

```
#include <fsprivate.h>

int write_page(void *buf, ui32 addr, ui32 type, ui32 wear, void *vol);
```

DESCRIPTION

write_page() is a driver callback function used by TargetFFS-NAND to write the flash memory page at address ***addr***. The size of a page is set by the "page_size" value in the FfsVol structure passed by **FsNandAddVol()**. ***buf*** is the address of the buffer containing the data to be written. This buffer is guaranteed to be aligned on a four-byte boundary.

wear is the wear count for the block containing the addressed page. For NAND devices which guarantee zero bit errors below a specified wear value, ***wear*** may be used to omit calculation of an error correction code (ECC), for higher performance.

type is a flag used by TargetFFS-NAND to mark the page as containing either user data or TargetFFS-NAND control data. It should be written to the page's extra bytes. No ECC encoding or decoding is performed on the type flag.

vol can be used to support multiple volumes. The value assigned to the "vol" field in the FfsVol structure that is passed to TargetFFS-NAND by **FsNandAddVol()** is passed unmodified as the last parameter in the flash driver callback functions. This field may be initialized to point to a structure containing volume-specific information, allowing one set of driver callback functions to support multiple volumes.

Error correction is done using an error correction code (ECC) written to the page's extra bytes. The ECC is read from the extra bytes by **read_page()**. TargetFFS-NAND supplies two fast ECC routines, **EncodeEcc()** for use in **write_page()** and **DecodeEcc()** for use in **read_page()**. Alternately, if available, hardware ECC support may be used.

The ECC bytes and the 32-bit ***type*** flag stored in the page's extra bytes may follow any format as long as **write_page()**, **read_page()**, and **write_type()** all use the same organization. The example driver in the next chapter uses the layout below.

Offset	Value
0-9	EncodeEcc() bytes 0-9
10-13	***type*** in big-endian order

write_page() should not return until the write is complete. It can poll a flash device's “ready” bit or, on systems that support this, block until an interrupt from the flash device signals that the operation is complete.

To detect "program disturb" errors, where '1's are inadvertently set to '0's, **write_page()** is required to read-verify the data, the type flag, and the ECC values it programs, to ensure there is an exact match with what was written.

write_page() returns 0 for success, 1 if read-verification fails, or -1 if a chip error occurs. If -1 is returned, the block is marked "bad" by TargetFFS-NAND and bad block recovery is performed. Once a block is marked bad, it is no longer erased or written.

write_type()

PROTOTYPE

```
#include <fsprivate.h>

int write_type(ui32 addr, ui32 type, void *vol);
```

DESCRIPTION

write_type() is a driver callback function used by TargetFFS-NAND to modify the type flag for the flash page at address ***addr***, updating it to record that the page no longer contains valid TargetFFS-NAND control data.

Partial programming (programming the same page more than once between block erasures) is limited for NAND devices. The standard **write_type()** implementation requires partial programming. However, **write_type()** is never called on a page more than once between successive block erase commands and this is usually less than a device's partial programming limit.

If partial programming is not allowed, **write_type()** may be implemented as an empty function. The only drawback is that volume mounts will take longer. Multiple pages will be marked as containing valid control information and TargetFFS-NAND will have to read them all to find the latest copy.

type is the new type flag written into the page's extra bytes. No ECC encoding or decoding is performed on the type flag. Since ***type*** may contain '1's in positions where the previous flag had '0's, read-back verification cannot be used to ensure the value has been correctly written. It is satisfactory if the chip's status code indicates that there are no errors.

vol can be used to support multiple volumes. The value assigned to the "vol" field in the FfsVol structure that is passed to TargetFFS-NAND by **FsNandAddVol()** is passed unmodified as the last parameter in the flash driver callback functions. This field may be initialized to point to a structure containing volume-specific information, allowing one set of driver callback functions to support multiple volumes.

write_type() should not return until its writes are complete. It can poll a flash device's “ready” bit or, on systems that support this, block until an interrupt from the flash device signals that the operation is complete.

write_type() returns 0 for success or -1 if a chip error occurs. If -1 is returned, the block is marked "bad" by TargetFFS-NAND and bad block recovery is performed. Once a block is marked bad, it is no longer erased or written to.

Sample Driver 5

Introduction

This chapter contains a sample TargetFFS-NAND driver for an 8MB device with an 8-bit interface. This driver can be used with most NAND devices simply by modifying the BLOCK_SIZE and NUM_BLOCKS definitions and by supplying equivalent code to read the RDY input and control the ALE, CE, CLE, and WP outputs. ECC encoding and decoding is done by calling the TargetFFS-NAND EncodeEcc() and DecodeEcc() routines, but these calls could be substituted with accesses to special registers if hardware ECC support is provided.

8MB Volume with 8-bit Interface

```
/***********************************************************************/
/*                                                                     */
/*   Module:  nanddrvr.c                                               */
/*   Release: 2003.2                                                   */
/*   Version: 2003.3                                                   */
/*   Purpose: NAND Flash File System Driver                            */
/*                                                                     */
/*---------------------------------------------------------------------*/
/*                                                                     */
/*               Copyright 2003, Blunk Microsystems                    */
/*                      ALL RIGHTS RESERVED                            */
/*                                                                     */
/*   Licensees have the non-exclusive right to use, modify, or extract */
/*   this computer program for software development at a single site.  */
/*   This program may be resold or disseminated in executable format   */
/*   only. The source code may not be redistributed or resold.         */
/*                                                                     */
/***********************************************************************/
#include "../posix.h"
#include "../include/libc/string.h"
#include "../include/targetos.h"
#include "../include/sys.h"
#include "../include/fsprivate.h"

#include "xp860t.h"
#include "mpc860.h"

/***********************************************************************/
/* Symbol Definitions                                                  */
/***********************************************************************/
#define BLOCK_SIZE      (8 * 1024)
#define NUM_BLOCKS      1024
#define PAGE_SIZE       512
```

```
/***********************************************************************/
/* Global Variable Definitions                                         */
/***********************************************************************/
static int ExtraBytes;  // TRUE iff chip is setup to access extra bytes

/***********************************************************************/
/* Function Prototypes                                                 */
/***********************************************************************/
void perror(const char *s);
void SysWait50ms(void);

/***********************************************************************/
/* Local Function Definitions                                          */
/***********************************************************************/

/***********************************************************************/
/* page_erased: Check if page's data bytes and extra bytes are erased  */
/*                                                                     */
/*      Inputs: addr = the page address                                */
/*              vol = driver's volume pointer                          */
/*                                                                     */
/*     Returns: TRUE if erased, FALSE if any bit is not 1              */
/*                                                                     */
/***********************************************************************/
static int page_erased(ui32 addr, void *vol)
{
  int n = PAGE_SIZE + 16;

  /*-----------------------------------------------------------------*/
  /* Clear any pre-existing interrupt and begin access.              */
  /*-----------------------------------------------------------------*/
  MPC860_SIPEND = IRQ3;
  lowerCE();

  /*-----------------------------------------------------------------*/
  /* Send command to start reading at first half of NAND data page.  */
  /*-----------------------------------------------------------------*/
  raiseCLE();
  NandPort = 0x00;  /* read command */
  ExtraBytes = FALSE;
  lowerCLE();

  /*-----------------------------------------------------------------*/
  /* Send address as three separate bytes.                           */
  /*-----------------------------------------------------------------*/
  raiseALE();
  NandPort = 0;
  NandPort = (ui8)(addr >> 9);
  NandPort = (ui8)(addr >> 17);
  lowerALE();

  /*-----------------------------------------------------------------*/
  /* Wait until device is ready.                                     */
  /*-----------------------------------------------------------------*/
  while ((MPC860_SIPEND & IRQ3) == FALSE) ;
  /*-----------------------------------------------------------------*/
  /* Check one byte at a time.                                       */
  /*-----------------------------------------------------------------*/
  do
    if (NandPort != 0xFF)
    {
      raiseCE();
      return FALSE;
    }
  while (--n);

  /*-----------------------------------------------------------------*/
  /* End device access and return success.                           */
  /*-----------------------------------------------------------------*/
  raiseCE();
  return TRUE;
}

/*******************************************************************/
/*  write_page: ECC encode and write one NAND flash page           */
/*                                                                 */
/*      Inputs: buffer = pointer to buffer containing data to write    */
/*              type = page type flag                              */
/*              addr = the page address                            */
/*              wear = wear count of this page's block             */
/*              vol = driver's volume pointer                      */
/*                                                                 */
/*     Returns: 0 for success, -1 for chip error, 1 for verify error   */
/*                                                                 */
/*******************************************************************/
static int write_page(void *buffer, ui32 addr, ui32 type, ui32 wear,
                      void *vol)
{
  int n;
  ui8 *dst, ecc[10], status;

  PfAssert(page_erased(addr, vol));

  /*-----------------------------------------------------------------*/
  /* Ensure the NAND address pointer is back to the data page.       */
  /*-----------------------------------------------------------------*/
  if (ExtraBytes)
  {
    /*---------------------------------------------------------------*/
    /* Clear any pre-existing interrupt and begin access.            */
    /*---------------------------------------------------------------*/
    MPC860_SIPEND = IRQ3;
    lowerCE();

    /*---------------------------------------------------------------*/
    /* Send command to start reading at first half of NAND data page.  */
    /*---------------------------------------------------------------*/
    raiseCLE();
    NandPort = 0x00;  /* read command */
    lowerCLE();

    /*---------------------------------------------------------------*/
    /* Send address 0 as three separate bytes.                         */
    /*-----------------------------------------------------------------*/
    raiseALE();
    NandPort = 0;
    NandPort = 0;
    NandPort = 0;
    lowerALE();

    /*-----------------------------------------------------------------*/
    /* Wait until device is ready.                                     */
    /*-----------------------------------------------------------------*/
    while ((MPC860_SIPEND & IRQ3) == FALSE) ;

    /*-----------------------------------------------------------------*/
    /* Read one byte and then end the device access.                   */
    /*-----------------------------------------------------------------*/
    NandPort;
    raiseCE();

    /*-----------------------------------------------------------------*/
    /* Clear flag since NAND pointer is reset to start of data page.   */
    /*-----------------------------------------------------------------*/
    ExtraBytes = FALSE;
  }

  /*-------------------------------------------------------------------*/
  /* Compute page's ECC encoding.                                      */
  /*-------------------------------------------------------------------*/
  EncodeEcc(buffer, ecc);

  /*-------------------------------------------------------------------*/
  /* Clear any pre-existing interrupt and begin write access.          */
  /*-------------------------------------------------------------------*/
  MPC860_SIPEND = IRQ3;
  lowerCE();

  /*-------------------------------------------------------------------*/
  /* Send Serial Data Input command.                                   */
  /*-------------------------------------------------------------------*/
  raiseCLE();
  NandPort = 0x80;  /* serial data input command */
  lowerCLE();

  /*-------------------------------------------------------------------*/
  /* Send address as three separate bytes.                             */
  /*-------------------------------------------------------------------*/
  raiseALE();
  NandPort = 0;
  NandPort = (ui8)(addr >> 9);
  NandPort = (ui8)(addr >> 17);
  lowerALE();

  /*-------------------------------------------------------------------*/
  /* Write the data portion.                                           */
  /*-------------------------------------------------------------------*/
  for (dst = buffer, n = 0; n < PAGE_SIZE; ++n)
    NandPort = *dst++;
```

```
/*-------------------------------------------------------------------*/
/* Write the 10-byte ECC portion.                                    */
/*-------------------------------------------------------------------*/
for (n = 0; n < 10; ++n)
  NandPort = ecc[n];

/*-------------------------------------------------------------------*/
/* Write the four type bytes.                                        */
/*-------------------------------------------------------------------*/
NandPort = (ui8)(type >> 24);
NandPort = (ui8)(type >> 16);
NandPort = (ui8)(type >> 8);
NandPort = (ui8)type;

/*-------------------------------------------------------------------*/
/* Send Auto Program command.                                        */
/*-------------------------------------------------------------------*/
raiseCLE();
NandPort = 0x10;  /* auto program command */
lowerCLE();

/*-------------------------------------------------------------------*/
/* Wait until device is ready.                                       */
/*-------------------------------------------------------------------*/
while ((MPC860_SIPEND & IRQ3) == FALSE) ;

/*-------------------------------------------------------------------*/
/* Send Status Read command.                                         */
/*-------------------------------------------------------------------*/
raiseCLE();
NandPort = 0x70;  /* status read command */
lowerCLE();

/*-------------------------------------------------------------------*/
/* Read device status and end write access.                          */
/*-------------------------------------------------------------------*/
status = NandPort;
raiseCE();

/*-------------------------------------------------------------------*/
/* Return -1 if chip reports program error.                          */
/*-------------------------------------------------------------------*/
if (status != 0xC0)
  return -1;

/*-------------------------------------------------------------------*/
/* Clear any previous interrupt and begin read-verify access.        */
/*-------------------------------------------------------------------*/
MPC860_SIPEND = IRQ3;
lowerCE();

/*-------------------------------------------------------------------*/
/* Send command to start reading at first half of NAND data page.    */
/*-------------------------------------------------------------------*/
raiseCLE();
NandPort = 0x00;  /* read command */
```

```
  lowerCLE();

  /*-------------------------------------------------------------------*/
  /* Send address as three separate bytes.                             */
  /*-------------------------------------------------------------------*/
  raiseALE();
  NandPort = 0;
  NandPort = (ui8)(addr >> 9);
  NandPort = (ui8)(addr >> 17);
  lowerALE();

  /*-------------------------------------------------------------------*/
  /* Wait until device is ready.                                       */
  /*-------------------------------------------------------------------*/
  while ((MPC860_SIPEND & IRQ3) == FALSE) ;

  /*-------------------------------------------------------------------*/
  /* Read-verify the data portion.                                     */
  /*-------------------------------------------------------------------*/
  for (dst = buffer, n = 0; n < PAGE_SIZE; ++n)
    if (*dst++ != NandPort)
    {
      raiseCE();
      return 1;
    }

  /*-------------------------------------------------------------------*/
  /* Read-verify the 10-byte ECC portion.                              */
  /*-------------------------------------------------------------------*/
  for (n = 0; n < 10; ++n)
    if (NandPort != ecc[n])
    {
      raiseCE();
      return 1;
    }

  /*-------------------------------------------------------------------*/
  /* End device access and return success.                             */
  /*-------------------------------------------------------------------*/
  raiseCE();
  return 0;
}

/*********************************************************************/
/*  write_type: Write page's type value                              */
/*                                                                   */
/*      Inputs: addr = the page address                              */
/*              type = page type (0xFFFF0000)                        */
/*              vol = driver's volume pointer                        */
/*                                                                   */
/*        Note: Updates four-byte type flag written by wr_page()     */
/*                                                                   */
/*********************************************************************/
static int write_type(ui32 addr, ui32 type, void *vol)
{
  ui8 status;
```

```
/*-----------------------------------------------------------------*/
/* Ensure the NAND address pointer is set to the extra bytes.      */
/*-----------------------------------------------------------------*/
if (ExtraBytes == FALSE)
{
  /*---------------------------------------------------------------*/
  /* Clear any pre-existing interrupt and begin access.            */
  /*---------------------------------------------------------------*/
  MPC860_SIPEND = IRQ3;
  lowerCE();

  /*---------------------------------------------------------------*/
  /* Send command to start reading at the NAND extra bytes.        */
  /*---------------------------------------------------------------*/
  raiseCLE();
  NandPort = 0x50;  /* read command */
  lowerCLE();

  /*---------------------------------------------------------------*/
  /* Send address 0 as three separate bytes.                       */
  /*---------------------------------------------------------------*/
  raiseALE();
  NandPort = 0;
  NandPort = 0;
  NandPort = 0;
  lowerALE();

  /*---------------------------------------------------------------*/
  /* Wait until device is ready.                                   */
  /*---------------------------------------------------------------*/
  while ((MPC860_SIPEND & IRQ3) == FALSE) ;

  /*---------------------------------------------------------------*/
  /* Read one byte and then end the device access.                 */
  /*---------------------------------------------------------------*/
  NandPort;
  raiseCE();

  /*---------------------------------------------------------------*/
  /* Set flag since NAND pointer is set to the extra bytes.        */
  /*---------------------------------------------------------------*/
  ExtraBytes = TRUE;
}

/*-----------------------------------------------------------------*/
/* Clear any pre-existing interrupt and begin write access.        */
/*-----------------------------------------------------------------*/
MPC860_SIPEND = IRQ3;
lowerCE();

/*-----------------------------------------------------------------*/
/* Send Serial Data Input command.                                 */
/*-----------------------------------------------------------------*/
raiseCLE();
NandPort = 0x80;  /* serial data input command */
lowerCLE();
```

```
    /*--------------------------------------------------------------------*/
    /* Send address as three separate bytes.                              */
    /*--------------------------------------------------------------------*/
    raiseALE();
    NandPort = 10;    /* start just past ECC bytes */
    NandPort = (ui8)(addr >> 9);
    NandPort = (ui8)(addr >> 17);
    lowerALE();

    /*--------------------------------------------------------------------*/
    /* Write the four type bytes.                                         */
    /*--------------------------------------------------------------------*/
    NandPort = (ui8)(type >> 24);
    NandPort = (ui8)(type >> 16);
    NandPort = (ui8)(type >> 8);
    NandPort = (ui8)type;

    /*--------------------------------------------------------------------*/
    /* Send Auto Program command.                                         */
    /*--------------------------------------------------------------------*/
    raiseCLE();
    NandPort = 0x10;  /* auto program command */
    lowerCLE();

    /*--------------------------------------------------------------------*/
    /* Wait until device is ready.                                        */
    /*--------------------------------------------------------------------*/
    while ((MPC860_SIPEND & IRQ3) == FALSE) ;

    /*--------------------------------------------------------------------*/
    /* Send Status Read command.                                          */
    /*--------------------------------------------------------------------*/
    raiseCLE();
    NandPort = 0x70;  /* status read command */
    lowerCLE();

    /*--------------------------------------------------------------------*/
    /* Read device status and end write access.                           */
    /*--------------------------------------------------------------------*/
    status = NandPort;
    raiseCE();

    /*--------------------------------------------------------------------*/
    /* Return -1 if chip reports program error. Else return success.     */
    /*--------------------------------------------------------------------*/
    if (status != 0xC0)
      return -1;
    else
      return 0;
  }

  /**********************************************************************/
  /*   read_type: Read page's write_page() type value                   */
  /*                                                                    */
  /*      Inputs: addr = the page address                               */
  /*              vol = driver's volume pointer                         */
  /*                                                                    */
```

```
/*     Returns: The four byte type value written during write_page()   */
/*              and possibly updated by write_type().                  */
/*                                                                     */
/***********************************************************************/
static ui32 read_type(ui32 addr, void *vol)
{
  ui32 type;

  /*-----------------------------------------------------------------*/
  /* Clear any pre-existing interrupt and begin access.              */
  /*-----------------------------------------------------------------*/
  MPC860_SIPEND = IRQ3;
  lowerCE();

  /*-----------------------------------------------------------------*/
  /* Send read extra bytes command and set flag.                     */
  /*-----------------------------------------------------------------*/
  raiseCLE();
  NandPort = 0x50;  /* read extra bytes command */
  ExtraBytes = TRUE;
  lowerCLE();

  /*-----------------------------------------------------------------*/
  /* Send address as three separate bytes.                           */
  /*-----------------------------------------------------------------*/
  raiseALE();
  NandPort = 10;    /* start just past ECC bytes */
  NandPort = (ui8)(addr >> 9);
  NandPort = (ui8)(addr >> 17);
  lowerALE();

  /*-----------------------------------------------------------------*/
  /* Wait until device is ready.                                     */
  /*-----------------------------------------------------------------*/
  while ((MPC860_SIPEND & IRQ3) == FALSE) ;

  /*-----------------------------------------------------------------*/
  /* Read the four type bytes.                                       */
  /*-----------------------------------------------------------------*/
  type = NandPort;
  type = (type << 8) | NandPort;
  type = (type << 8) | NandPort;
  type = (type << 8) | NandPort;

  /*-----------------------------------------------------------------*/
  /* End the device access and return the coded type value.          */
  /*-----------------------------------------------------------------*/
  raiseCE();
  return type;
}

/***********************************************************************/
/*   read_page: Read and ECC correct one NAND flash page               */
/*                                                                     */
/*      Inputs: buffer = pointer to buffer to copy data to             */
/*              addr = the page address                                */
/*              wear = wear count of this page's block                 */
/*              vol = driver's volume pointer                            */
/*                                                                       */
/*     Returns: 0 for success, -1 for uncorrectable error                */
/*                                                                       */
/*************************************************************************/
static int read_page(void *buffer, ui32 addr, ui32 wear, void *vol)
{
  int n;
  ui8 *dst = buffer;
  ui8 ecc[10];

  /*---------------------------------------------------------------------*/
  /* Clear any pre-existing interrupt and begin access.                  */
  /*---------------------------------------------------------------------*/
  MPC860_SIPEND = IRQ3;
  lowerCE();

  /*---------------------------------------------------------------------*/
  /* Send command to start reading at first half of NAND data page.      */
  /*---------------------------------------------------------------------*/
  raiseCLE();
  NandPort = 0x00;  /* read command */
  ExtraBytes = FALSE;
  lowerCLE();

  /*---------------------------------------------------------------------*/
  /* Send address as three separate bytes.                               */
  /*---------------------------------------------------------------------*/
  raiseALE();
  NandPort = 0;
  NandPort = (ui8)(addr >> 9);
  NandPort = (ui8)(addr >> 17);
  lowerALE();

  /*---------------------------------------------------------------------*/
  /* Wait until device is ready.                                         */
  /*---------------------------------------------------------------------*/
  while ((MPC860_SIPEND & IRQ3) == FALSE) ;

  /*---------------------------------------------------------------------*/
  /* Read the data portion.                                              */
  /*---------------------------------------------------------------------*/
  for (n = 0; n < PAGE_SIZE; ++n)
    *dst++ = NandPort;

  /*---------------------------------------------------------------------*/
  /* Read the 10-byte ECC portion.                                       */
  /*---------------------------------------------------------------------*/
  for (n = 0; n < 10; ++n)
    ecc[n] = NandPort;

  /*---------------------------------------------------------------------*/
  /* End the device access.                                              */
  /*---------------------------------------------------------------------*/
  raiseCE();

  /*---------------------------------------------------------------------*/
  /* Perform ECC decoding, return error if uncorrectable error.        */
  /*-----------------------------------------------------------------*/
  if (DecodeEcc(buffer, ecc))
    return -1;
  else
    return 0;
}

/*******************************************************************/
/* erase_block: Erase entire flash block                             */
/*                                                                   */
/*      Inputs: addr = base address of block to be erased            */
/*              vol = driver's volume pointer                        */
/*                                                                   */
/*     Returns: 0 on success, -1 on failure                          */
/*                                                                   */
/*******************************************************************/
static int erase_block(ui32 addr, void *vol)
{
  ui8 status;

  /*-----------------------------------------------------------------*/
  /* Clear any pre-existing interrupt and begin access.              */
  /*-----------------------------------------------------------------*/
  MPC860_SIPEND = IRQ3;
  lowerCE();

  /*-----------------------------------------------------------------*/
  /* Send Auto Block Erase Setup command.                            */
  /*-----------------------------------------------------------------*/
  raiseCLE();
  NandPort = 0x60;  /* auto block erase setup command */
  lowerCLE();

  /*-----------------------------------------------------------------*/
  /* Send address as two separate bytes.                             */
  /*-----------------------------------------------------------------*/
  raiseALE();
  NandPort = (ui8)(addr >> 9);
  NandPort = (ui8)(addr >> 17);
  lowerALE();

  /*-----------------------------------------------------------------*/
  /* Send Erase Start command.                                       */
  /*-----------------------------------------------------------------*/
  raiseCLE();
  NandPort = 0xD0;  /* erase start command */
  lowerCLE();

  /*-----------------------------------------------------------------*/
  /* Wait until device is ready.                                     */
  /*-----------------------------------------------------------------*/
  while ((MPC860_SIPEND & IRQ3) == FALSE) ;

  /*-----------------------------------------------------------------*/
  /* Send Status Read command.                                       */
  /*-----------------------------------------------------------------*/
  raiseCLE();
  NandPort = 0x70;  /* status read command */
  lowerCLE();

  /*-----------------------------------------------------------------*/
  /* Read device status.                                             */
  /*-----------------------------------------------------------------*/
  status = NandPort;

  /*-----------------------------------------------------------------*/
  /* End device access and return success.                           */
  /*-----------------------------------------------------------------*/
  raiseCE();
  return status == 0xC0 ? 0 : -1;
}

/*********************************************************************/
/* Global Function Definitions                                       */
/*********************************************************************/

/*********************************************************************/
/* NandDriverModule: Flash driver's module list entry                */
/*                                                                   */
/*      Inputs: req = module request code                            */
/*              ... = additional parameters specific to request      */
/*                                                                   */
/*********************************************************************/
void *FlashDriverModule(int req, ...)
{
  FfsVol vol;

  if (req == kInitMod)
  {
    /*---------------------------------------------------------------*/
    /* Reset the NAND device.                                        */
    /*---------------------------------------------------------------*/
    lowerCE();
    raiseCLE();
    NandPort = 0xFF;    /* reset command */
    lowerCLE();
    raiseCE();

    /*---------------------------------------------------------------*/
    /* Wait until the device is ready.                               */
    /*---------------------------------------------------------------*/
    MPC860_SIEL &= ~ED3; /* -> level mode */
    SysWait50ms();
    while ((MPC860_SIPEND & IRQ3) == FALSE) ;
    MPC860_SIEL |= ED3;  /* -> edge mode */

    /*---------------------------------------------------------------*/
    /* Remove write protection, allowing the device to be programmed.  */
    /*---------------------------------------------------------------*/
    raiseWP();

    /*---------------------------------------------------------------*/
    /* Register flash NAND volume with TargetFFS.                    */
    /*-----------------------------------------------------------------*/
    vol.name = "flash";
    vol.page_size = PAGE_SIZE;
    vol.block_size = BLOCK_SIZE;
    vol.num_blocks = NUM_BLOCKS;
    vol.max_bad_blocks = 10;   /* from data sheet */
    vol.mem_base = 0;
    vol.vol = NULL;
    vol.driver.nand.write_page = write_page;
    vol.driver.nand.write_type = write_type;
    vol.driver.nand.read_page = read_page;
    vol.driver.nand.read_type = read_type;
    vol.driver.nand.page_erased = page_erased;
    vol.driver.nand.erase_block = erase_block;
    if (FsNandAddVol(&vol))
      perror("FsNandAddVol() failed");
  }

  return NULL;
}
```

Intentionally left blank.

Appendix A – Error Codes

If an error occurs, the respective call sets errno to a non-zero error code and returns -1 or NULL. The error codes generated by TargetOS file systems are listed below. If there is an error in a lower level driver or protocol layer, errno will be set to the appropriate driver or protocol error code.

EPERM (1)	The user ID determined by **FsGetId()** does not match the owner of the file.
ENOENT (2)	The specified file or directory does not exist.
EIO (5)	Input or output error.
ENXIO (6)	The specified device does not exist or is not ready.
EBADF (9)	The specified file descriptor is invalid.
ENOMEM (12)	Out of memory.
EACCES (13)	Search permission is denied for a directory in the specified path or an unpermitted operation was attempted.
EFAULT (14)	Bad address.
EBUSY (16)	The specified file is either open or the current working directory of another task.
EEXIST (17)	The named file already exists.
EXDEV (18)	Improper link, attempted link to another file system.
ENOTDIR (20)	A component of the specified path is not a directory.
EISDIR (21)	Attempted to open a directory for writing or to rename a file to an existing directory name.
EINVAL (22)	An argument value is invalid.
EMFILE (24)	Too many files are currently open.
ENOTTY (25)	Not a terminal device.
ENOSPC (28)	The specified file system volume is full.

EROFS (30)	The file system is read-only.
ENAMETOOLONG (38)	The length of the specified file name exceeds PATH_MAX.

Appendix B – Standard C Format Specifiers

The portion of the Standard C specification regarding format specifiers for fprintf(), fscanf(), and their related functions has been reproduced, with permission, in this appendix.

7.9.6.1 The fprintf function

Synopsis

```
#include <stdio.h>
int fprintf(FILE *stream, const *format, ...);
```

Description

The **fprintf** function writes output to the stream pointed to by **stream**, under control of the string pointed to by **format** that specifies how subsequent arguments are converted for output. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated (as always) but are otherwise ignored. The **fprintf** function returns when the end of the format string is encountered.

The format shall be a multibyte character sequence, beginning and ending in its initial shift state. The format is composed of zero or more directives; ordinary multibyte characters (not **%**), which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments. Each conversion specification is introduced by the character **%**. After the **%**, the following appear in sequence:

— Zero or more *flags* (in any order) that modify the meaning of the conversion specification.

— An optional minimum *field width*. If the converted value has fewer characters than the field width, it will be padded with spaces (by default) on the left (or right, if the left adjustment flag, described later, has been given) to the field width. The field width takes the form of an asterisk ***** (described later) or a decimal integer.[118]

— An optional *precision* that gives the minimum number of digits to appear for the **d**, **i**, **o**, **u**, **x**, and **X** conversion, the number of digits to appear after the decimal-point character for **e**, **E**, and **f** conversions, the maximum number of significant digits for the **g** and **G** conversions, or the maximum number of characters to be written from a string in **s** conversion. The precision takes the form of a period (**.**) followed either by an asterisk ***** (described later) or by an optional decimal integer; if only the period is specified, the precision is taken as zero. If a precision appears with any other conversion specifier, the behavior is undefined.

— An optional **h** specifying that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to a **short int** or **unsigned short int** argument (the argument will have been promoted according to the integral promotions, and its value shall be converted to **short int** or **unsigned short int** before printing); an optional **h** specifying that a following **n** conversion specifier applies to a pointer to a **short int** argument; and optional **l** (ell) specifying that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to a **long int** or **unsigned long int** argument; an optional **l** specifying that a following **n** conversion specifier applies to a pointer to a **long int** argument; or an optional **L** specifying that a following **e**, **E**, **f**, **g**, or **G** conversion specifier applies to a **long double** argument. If an **h**, **l**, or **L** appears with any other conversion specifier, the behavior is undefined.

— A character that specifies the type of conversion to be applies.

As noted above, a field width, or precision, or both, may be indicated by an asterisk. In this case, an **int** argument supplies the field width or precision. The arguments specifying field width, or precision, or both, shall appear (in that order) before the argument (if any) to be converted. A negative field width argument is taken as a **-** flag followed by a positive field width. A negative precision argument is taken as if the precision were omitted.

The flag characters and their meanings are

- The result of the conversion will be left-justified within the field. (It will be right-justified if this flag is not specified.)

+ The result of a signed conversion will always begin with a plus or minus sign. (It will begin with a sign only when a negative value is converted if this flag is not specified.)

space If the first character of a signed conversion is not a sign, or if a signed conversion results in no characters, a space will be prefixed to the result. If the *space* and **+** flags both appear, the *space* flag will be ignored.

# The result is to be converted to an "alternate form." For **o** conversion, it increases the precision to force the first digit of the result to be a zero. For **x** (or **X**) conversion, a nonzero result will have **0x** (or **0X**) prefixed to it. For **e**, **E**, **f**, **g**, and **G** conversions, the result will always contain a decimal-point character, even if no digits follow it. (Normally, a decimal-point character appears in the result of these conversions only if a digit follows it.) For **g**

[118] Note that 0 is taken as a flag, not as the beginning of a field width.

and **G** conversions, trailing zeros will *not* be removed from the result. For other conversions, the behavior is undefined.

0 For **d**, **i**, **o**, **u**, **x**, **X**, **e**, **E**, **f**, **g**, and **G** conversions, leading zeros (following any indication of sign or base) are used to pad to the field width; no space padding is performed. If the **0** and **-** flags both appear, the **0** flag will be ignored. For **d**, **i**, **o**, **u**, **x**, and **X** conversions, if a precision is specified, the **0** flag will be ignored. For other conversions, the behavior is undefined.

The conversion specifiers and their meanings are

d, i The **int** argument is converted to signed decimal in the style *[—]dddd*. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no characters.

o, u, x, X The **unsigned int** argument is converted to unsigned octal (**o**), unsigned decimal (**u**), or unsigned hexadecimal notation (**x** or **X**) in the style *dddd*; the letters **abcdef** are used for **x** conversion and the letters **ABCDEF** for **X** conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no characters.

f The **double** argument is converted to decimal notation in the style *[—]ddd.ddd*, where the number of digits after the decimal-point character is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is zero and the # flag is not specified. no decimal-point character appears. If a decimal-point character appears, at least one digit appears before it. The value is rounded to the appropriate number of digits.

e, E The **double** argument is converted in the style *[—]d.ddd***e**±*dd*, where there is one digit before the decimal-point character (which is nonzero if the argument is nonzero) and the number of digits after it is equal to the precision; if the precision is missing, it is taken as 6; if the precision is zero and the # flag is not specified, no decimal-point character appears. The value is rounded to the appropriate number of digits. The **E** conversion specifier will produce a number with **E** instead of **e** introducing the exponent. The exponent always contains at least two digits. If the value is zero, the exponent is zero.

g, G The **double** argument is convened in style **f** or **e** (or in style **E** in the case of a **G** conversion specifier), with the precision specifying the number of significant digits. If the precision is zero, it is taken as 1. The style used depends on the value converted; style **e** (or **E**) will be used only if the exponent resulting from such a conversion is less than -4 or greater than or equal to the precision. Trailing zeros are removed from the fractional portion of the result; a decimal-point character appears only if it is followed by a digit.

c The **int** argument is converted to an **unsigned char**, and the resulting character is written.

a The argument shall be a pointer to an array of character type.[119] Characters from the array are written up to (but not including) a terminating null character; if the precision is specified, no more than that many characters are written. If the precision is not specified or is greater than the size of the array, the array shall contain a null character.

[119] No special provisions are made for multibyte characters.

p The argument shall be a pointer to **void**. The value of the pointer is converted to a sequence of printable characters, in an implementation-defined manner.

n The argument shall be a pointer to an integer into which is *written* the number of characters written to the output stream so far by this call to **fprintf**. No argument is converted.

% A **%** is written. No argument is converted. The complete conversion specification shall be **%%.**

If a conversion specification is invalid, the behavior is undefined.[120]

If any argument is, or points to, a union or an aggregate (except for an array of character type using **%s** conversion, or a pointer using **%p** conversion), the behavior is undefined.

In no case does a nonexistent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is expanded to contain the conversion result.

Returns

The **fprintf** function returns the number of characters transmitted, or a negative value if an output error occurred.

Environmental limit

The minimum value for the maximum number of characters produced by any single conversion shall be 509.

Example

To print a date and time in the form “Sunday, July 3, 10:02” followed by **p** to five decimal places:

```
#include <math.h>
#include <stdio.h>
/*...*/
char *weekday, *month;      /* pointers to strings */
int day, hour, min;
fprintf(stdout, “%s, %s %d, %.2d:%.2d\n”,
        weekday, month, day, hour, min);
fprintf(stdout, “pi = %.5f\n”, 4 * atan(l.0));
```

7.9.6.2 The fscanf function

Synopsis

```
#include <stdio.h>
int fscanf (FILE *stream, const char *format, ...);
```

Description

The **fscanf** function reads input from the stream pointed to by **stream**, under control of the

[120] See “future library directions” (7.13.6).

string pointed to by **format** that specifies the admissible input sequences and how they are to be converted for assignment, using subsequent arguments as pointers to the objects to receive the converted input. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated (as always) but are otherwise ignored.

The format shall be a multibyte character sequence, beginning and ending in its initial shift state. The format is composed of zero or more directives; one or more white-space characters; an ordinary multibyte character (neither **%** nor a white-space character); or a conversion specification. Each conversion specification is introduced by the character **%**. After the **%**, the following appear in sequence:

— An optional assignment-suppressing character ***.**

— An optional nonzero decimal integer that specifies the maximum field width.

— An optional **h**, **l** (ell) or **L** indicating the size of the receiving object. The conversion specifiers **d**, **i**, and **n** shall be preceded by **h** if the corresponding argument is a pointer to **short int** rather than a pointer to **int**, or by **1** if it is a pointer to **long int**. Similarly, the conversion specifiers **o**, **u**, and **x** shall be preceded by **h** if the corresponding argument is a pointer to **unsigned short int** rather than a pointer to **unsigned in,**. or by **1** if it is a pointer to **unsigned long int**. Finally, the conversion specifiers **e**, **f**, and **g** shall be preceded by **1** if the corresponding argument is a pointer to double rather than a pointer to **float**, or by **L** if it is a pointer to **long double**. If an **h**, **1**, or **L** appears with any other conversion specifier, the behavior is undefined.

— A character that specifies the type of conversion to be applied. The valid conversion specifiers are described below.

The **fscanf** function executes each directive of the format in turn. If a directive fails, as detailed below, the **fscanf** function returns. Failures are described as input failures (due to the unavailability of input characters), or matching failures (due to inappropriate input).

A directive composed of white-space character(s) is executed by reading input up to the first non-white-space character (which remains unread), or until no more characters can be read.

A directive that is an ordinary multibyte character is executed by reading the next characters of the stream. If one of the characters differs from one comprising the directive, the directive fails, and the differing and subsequent characters remain unread.

A directive that is a conversion specification defines a set of matching input sequences, as described below for each specifier. A conversion specification is executed in the following steps:

Input white-space characters (as specified by the **isspace** function) are skipped, unless the specification includes a **[**, **c**, or **n** specifier.[121]

An input item is read from the stream, unless the specification includes an **n** specifier. An input item is defined as the longest matching sequence of input characters, unless that exceeds a specified field width, in which case it is the initial subsequence of that length in the sequence. The first character, if any, after the input item remains unread. If the length of the input item is zero, the execution of the directive fails; this condition is a matching failure, unless an error prevented input from the stream, in which case it is an input failure.

Except in the case of a **%** specifier, the input item (or, in the case of a **%n** directive, the count of input characters) is converted to a type appropriate to the conversion specifier. If the input

[121] These white-space characters are not counted against a specified field width.

item is not a matching sequence, the execution of the directive fails; this condition is a matching failure. Unless assignment suppression was indicated by a *****, the result of the conversion is placed in the object pointed to by the first argument following the **format** argument that has not already received a conversion result. If this object does not have an appropriate type, or if the result of the conversion cannot be represented in the space provided, the behavior is undefined.

The following conversion specifiers are valid:

d Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the **strtol** function with the value 10 for the **base** argument. The corresponding argument shall be a pointer to integer.

i Matches an optionally signed integer, whose format is the same as expected for the subject sequence of the **strtol** function with the value 0 for the **base** argument. The corresponding argument shall be a pointer to integer.

o Matches an optionally signed octal integer, whose format is the same as expected for the subject sequence of the **strtoul** function with the value 8 for the **base** argument. The corresponding argument shall be a pointer to unsigned integer.

u Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the **strtoul** function with the value 10 for the **base** argument. The corresponding argument shall be a pointer to unsigned integer.

x Matches an optionally signed hexadecimal integer, whose format is the same as expected for the subject sequence of the **strtoul** function with the value 16 for the **base** argument. The corresponding argument shall be a pointer to unsigned integer.

e, f, g Matches an optionally signed floating-point number, whose format is the same as expected for the subject string of the **strtod** function. The corresponding argument shall be a pointer to floating.

s Matches a sequence of non-white-space characters.[122] The corresponding argument shall be a pointer to the initial character of an array large enough to accept the sequence and a terminating null character, which will be added automatically.

[Matches a nonempty sequence of characters[122] from a set of expected characters (the *scanset).* The corresponding argument shall be a pointer to the initial character of an array large enough to accept the sequence and a terminating null character, which will be added automatically. The conversion specifier includes all subsequent characters in the **format** string, up to and including the matching right bracket (**]**). The characters between the brackets (the *scanlist)* comprise the scanset, unless the character after the left bracket is a circumflex **(^),** in which case the scanset contains all characters that do not appear in the scanlist between the circumflex and the right bracket. If the conversion specifier begins with **[]** or **[^],** the right bracket character is in the scanlist and the next right bracket character is the matching right bracket that ends the specification; otherwise the first right bracket character is the one that ends the specification. If a **-** character is in the scanlist and is not the first, nor the second where the first character is a **^,** nor the last character, the behavior is implementation-defined.

c Matches a sequence of characters[122] of the number specified by the field width (1 if no field width is present in the directive). The corresponding argument shall be a pointer to the initial character of an array large enough to accept the sequence. No null character is added.

[122] No special provisions are made for multibyte characters.

p Matches an implementation-defined set of sequences, which should be the same as the set of sequences that may be produced by the **%p** conversion of the **fprintf** function. The corresponding argument shall be a pointer to a pointer to **void**. The interpretation of the input item is implementation-defined. If the input item is a value converted earlier during the same program execution, the pointer that results shall compare equal to that value; otherwise the behavior of the **%p** conversion is undefined.

n No input is consumed. The corresponding argument shall be a pointer to integer into which is to be written the number of characters read from the input stream so far by this call to the **fscanf** function. Execution of a **%n** directive does not increment the assignment count returned at the completion of execution of the **fscanf** function.

% Matches a single **%**; no conversion or assignment occurs. The complete conversion specification shall be **%%**.

If a conversion specification is invalid, the behavior is undefined.[123]

The conversion specifiers **E**, **G**, and **X** are also valid and behave the same as, respectively, **e**, **g**, and **x**.

If end-of-file is encountered during input, conversion is terminated. If end-of-file occurs before any characters matching the current directive have been read (other than leading white space, where permitted), execution of the current directive terminates with an input failure; otherwise, unless execution of the current directive is terminated with a matching failure, execution of the following directive (if any) is terminated with an input failure.

If conversion terminates on a conflicting input character, the offending input character is left unread in the input stream. Trailing white space (including new-line characters) is left unread unless matched by a directive. The success of literal matches and suppressed assignments is not directly determinable other than via the **%n** directive.

Returns

The **fscanf** function returns the value of the macro **EOF** if an input failure occurs before any conversion. Otherwise, the **fscanf** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

Examples

1. The call:

```
#include  <stdio.h>
 /* … */
int n, i; float x; char name[50];
n = fscanf(stdin, “%d%f%s”, &i, &x, name);
```

with the input line:

```
25 54.32E-1 thompson
```

will assign to *n* the value **3**, to *i* the value **25**, to *x* the value **5.432**, and *name* will contain **thompson\0**.

[123] See “future library directions” (7.13.6).

2. The call:

```
#include <stdio.h>
/* … */
int i; float x; char name[50];
fscanf(stdin, “%2d%f%*d %[01234567B9]”, &i, &x, name);
```

with input:

```
56789 0123 56a72
```

will assign to *i* the value **56** and to *x* the value **789.0**, will skip **0123**, and *name* will contain **56\0**. The next character read from the input stream will be **a**.

3. To accept repeatedly from **stdin** a quantity, a unit of measure and an item name:

```
#include <stdio.h>
/* … */
int count; float quant; char units[21], item[21];
while (!feof(stdin) && !ferror(stdin)) {
        count = fscanf(stdin, “%f%20s of %20s”,
                &quant, units, item);
        fscanf(stdin, “%*(^\l]”);
}
```

If the **stdin** stream contains the following lines:

```
2 quarts of oil
-12.8degrees Celsius
lots of luck
10.OLBS of
dirt
100ergs of energy
```

the execution of the above example will be analogous to the following assignments:

```
quant = 2; strcpy(units, “quarts”); strcpy(item, “oil”);
count = 3;
quant = -12.8; strcpy(units, “degrees”);
count = 2; /* “C” fails to match “o” */
count = 0; /* “l” fails to match “%f” */
quant = 10.0; strcpy(units, “LBS”); strcpy(item, “dirt”);
count = 3;
count = 0; /* “100e” fails to match “%f” */
count = EOF;
```

Forward references: the **strtod** function (7.10.1.4), the **strtol** function (7.10.1.5), the **strtoul** function (7.10.1.6).

7.13 Future library directions

7.13.6 Input/output <stdio.h>

Lowercase letters may be added to the conversion specifiers in **fprintf** and **fscanf**. Other characters may be used in extensions.

Intentionally left blank.

Appendix C – Porting Notes

TargetFFS-NAND is developed and maintained using TargetOS, Blunk Microsystems' real-time kernel, and has been hosted on pSOS$^+$, Nucleus, VxWorks, and other kernels. It also works without a kernel. The code is 100% ANSI C and is easy to port. This appendix discusses issues involved in porting TargetFFS-NAND.

Kernel Objects

TargetFFS-NAND uses semaphores to protect critical sections in multitasking environments. Two semaphores are used for the common file system layer shared by all TargetOS file systems. Additionally, one semaphore is used for each TargetFFS-NAND volume.

The easiest approach for porting is to write wrapper functions that implement the TargetOS semCreate(), semDelete(), semPost(), and semPend() functions using the RTOS being ported to, so that it "looks" like TargetOS to the TargetFFS code. The TargetOS manual pages for these calls are included at the end of this appendix.

Name Collision

When not used with the TargetOS runtime library, TargetFFS-NAND's Standard C API is "name-mangled" to prevent linker collision by appending "FFS" to each standard name. I.e. feof() becomes feofFFS(). The TargetFFS-NAND public header file "ffs_stdio.h" contains prototypes for the new names.

Alternately, the TargetFFS-NAND Standard C API may be omitted. Applications would be restricted to using just the routines prototyped in "posix.h". Except for rename(), the Standard C API is a subset of the POSIX API in functionality. rename() can be retained in a "name mangled" form to keep its functionality.

Memory Allocation

TargetFFS-NAND uses malloc() and free() to allocate memory for control structures and cache buffers during operation. The environment must support re-entrant malloc() and free() functions.

OsSecCount

TargetFFS-NAND uses a global variable, OsSecCount, to set file access and modification times. If supported by the system being ported to, OsSecCount should be initialized at startup using mktime() and thereafter incremented once a second. If this is not done, the access and modification times returned by stat() should be ignored.

errno

RTOS's frequently support task-specific errno values, while polling environments typically use a global variable. To minimize porting efforts, TargetFFS-NAND does not directly access errno. All accesses to errno are done using the following macros, defined in "fsprivate.h", which must be modified to support the errno mechanism used in your environment:

```
#define set_errno(e)    (*(int *)((unsigned long)RunningTask + 12) = (e))
#define get_errno()     (*(int *)((unsigned long)RunningTask + 12))
```

semCreate()

PROTOTYPE

```
#include <kernel.h>

SEM semCreate(const char name[8], int count, int mode);
```

DESCRIPTION

semCreate() is used to allocate and initialize a semaphore. ***name*** points to a string up to eight characters long used to name the semaphore. The name can be useful for debugging. Also, the semaphore name can be converted to an identifier using **semGetId()**. No check is made to ensure semaphore names are unique. If names are duplicated, **semGetId()** returns the identifier of the semaphore created first.

Semaphores maintain an internal counter. If positive or zero, the count indicates the number of tokens available. If negative, it indicates the number of tasks blocked on the semaphore, waiting for tokens to be posted by **semPost()**. ***count*** is the initial semaphore count. It must be either zero or a positive number.

mode is either OS_FIFO or OS_PRIORITY. It determines how a task is selected when a token is posted to a semaphore with waiting tasks. If the mode is OS_FIFO, the longest waiting task is made ready. Otherwise, the highest priority task is made ready. If the kernel is configured to be preemptive and any ready task has higher priority than the running task, the running task is preempted.

When a semaphore is created, memory is allocated from the kernel partition. If FIFO task queuing is selected, 32 bytes are allocated for the semaphore control block. If priority task queuing is selected, 320 bytes are allocated: 64 bytes for the control block and 256 bytes for the priority queue.

If successful, **semCreate()** returns a pointer to the semaphore control block. This identifier is used in subsequent semaphore-related service calls. If a nonfatal error occurs, ***errno*** is set and NULL is returned. It is a fatal error to call **semCreate()** from an interrupt service routine, with ***count*** negative, or with ***mode*** neither OS_FIFO nor OS_PRIORITY.

ERROR CODES

ENOMEM	**semCreate()** was unable to allocate the memory needed.

EXAMPLE

```
/*-----------------------------------------------------------------*/
/* Initializaton for TCP socket list.                              */
/*-----------------------------------------------------------------*/
TcpListSem = semCreate("tcp list", 1, OS_FIFO);
```

semDelete()

PROTOTYPE

```
#include <kernel.h>

void semDelete(SEM *semp);
```

DESCRIPTION

semDelete() is used to delete a semaphore, freeing the kernel buffer allocated for the semaphore control block. The control block size is 32 bytes in FIFO mode, 64 bytes in priority mode. If the semaphore was created in priority mode, the 256-byte buffer allocated for the priority queue is freed also. ***semp*** points to a semaphore identifier previously returned by **semCreate()**.

Any tasks blocked on the semaphore, waiting for a token, are put on the kernel's ready list. If the kernel is configured to be preemptive and any ready task has higher priority than the running task, the running task is preempted. When next scheduled to run, these tasks return from **semPend()** with the return value of -1 and with ***errno*** set to OS_OBJ_DELETED.

If successful, **semDelete()** assigns NULL to ***semp***. It is a fatal error to call **semDelete()** from an interrupt service routine or with an invalid semaphore identifier.

EXAMPLE

```
/***********************************************************************/
/*       close: Free channel resource and disable channel interrupts   */
/*                                                                     */
/***********************************************************************/
static int close(void *handle)
{
  ...

  /*-----------------------------------------------------------------*/
  /* Delete channel access semaphores.                                 */
  /*-----------------------------------------------------------------*/
  semDelete(&RxAccess);
  semDelete(&TxAccess);
  return 0;
}
```

semPend()

PROTOTYPE

```
#include <kernel.h>

int semPend(SEM sem, ui32 wait_opt);
```

DESCRIPTION

semPend() is used to get a token from a semaphore. ***sem*** is an identifier previously returned by **semCreate()**. If either a token is available, ***wait_opt*** is NO_WAIT (0), or the caller is an interrupt service routine, then **semPend()** returns immediately. A task will wait indefinitely if ***wait_opt*** is WAIT_FOREVER (-1). Otherwise, the wait is limited to ***wait_opt*** kernel ticks.

When a token is posted to a semaphore with multiple waiting tasks, the semaphore's mode determines which task is made ready. If the semaphore was created in OS_FIFO mode, the longest waiting task is selected. Otherwise, the highest priority task is selected. If the kernel is configured to be preemptive and any ready task has higher priority than the running task, the running task is preempted.

Semaphores are general-purpose tools, useful for both resource allocation and task synchronization. They have a small memory footprint and low processing overhead. Unless the priority inheritance of a mutex, the broadcasting or transitory behavior of a nexus, or the data passing ability of a queue is needed, a semaphore should be used.

If successful, **semPend()** returns 0. Otherwise, ***errno*** is set and -1 is returned. It is a fatal error to call **semPend()** with an invalid semaphore identifier.

ERROR CODES

OS_OBJ_DELETED	The semaphore was deleted while the calling task was waiting for a token.
OS_WOULD_BLOCK	Semaphore has no tokens and either ***wait_opt*** is NO_WAIT or caller is an ISR.

EXAMPLE

```
void *GetBuffer(void)
{
  LapbBuf *bufp;

  semPend(SemId, WAIT_FOREVER);
  bufp = FreeHead;
  FreeHead = FreeHead->next;
  return bufp;
}
```

semPost()

PROTOTYPE

```
#include <kernel.h>

int semPost(SEM sem);
```

DESCRIPTION

semPost() is used to post a semaphore token. ***sem*** is a semaphore identifier previously returned by **semCreate()**. If a token is posted to a semaphore with multiple waiting tasks, the semaphore's mode determines which task is made ready. If the semaphore was created in OS_FIFO mode, the longest waiting task is selected. Otherwise, the highest priority task is selected.

If the kernel is preemptive and the task that receives the token has a higher priority than the running task, then the running task is preempted. The kernel places the running task on the kernel's ready list and switches to the task made ready by the semaphore token post.

Semaphores are general-purpose tools, useful for both resource allocation and task synchronization. They have a small memory footprint and low processing overhead. Unless the priority inheritance of a mutex, the broadcasting or transitory behavior of a nexus, or the data passing ability of a queue is needed, a semaphore should be used.

If successful, **semPost()** returns 0. If ***sem*** is NULL, ***errno*** is set to OS_INVALID_ID and -1 is returned. (When a semaphore is deleted, **semDelete()** assigns NULL to the semaphore identifier). It is a fatal error if ***sem*** is neither a valid semaphore identifier nor NULL.

ERROR CODES

OS_INVALID_ID	***sem*** is NULL.

EXAMPLE

```
/**********************************************************************/
/*   RetBuffer: Returns one buffer to free buffer pool               */
/*                                                                    */
/*       Input: buf_ptr = pointer to buffer being returned            */
/*                                                                    */
/**********************************************************************/
void RetBuffer(BufType *buf_ptr)
{
  buf_ptr->next = FreeHead;
  FreeHead = buf_ptr;
  semPost(BufSem);
}
```

Intentionally left blank.

Appendix D – API List

This appendix lists the full TargetOS file system API and shows which routines are supported by which file systems. The limitations come from the nature of the file systems. For example, TargetZFS is a read-only file system and so lacks the calls that create or modify files.

API	FAT	FFS	RFS	ZFS
access()	•	•	•	•
chdir()	•	•	•	•
chmod()		•	•	•
chown()		•	•	•
clearerr()	•	•	•	•
close()	•	•	•	•
closedir()	•	•	•	•
creat()	•	•	•	
creatn()		•		
disable_sync()		•		
dup()	•	•	•	
dup2()	•	•	•	
enable_sync()		•		
fclose()	•	•	•	•
fcntl()	•	•	•	
fdopen()	•	•	•	•
feof()	•	•	•	•
ferror()	•	•	•	•
fflush()	•	•	•	
fgetc()	•	•	•	•
fgetpos()	•	•	•	•
fgets()	•	•	•	•
fileno()	•	•	•	•
fopen()	•	•	•	•
format()	•	•	•	
fprintf()	•	•	•	
fputc()	•	•	•	
fputs()	•	•	•	
fread()	•	•	•	•
freopen()	•	•	•	•
fscanf()	•	•	•	•
fseek()	•	•	•	•
fsetpos()	•	•	•	•
fstat()	•	•	•	•
ftell()	•	•	•	•
ftruncate()	•	•	•	
fwrite()	•	•	•	
getc()	•	•	•	•
getchar()	•	•	•	•
getcwd()	•	•	•	•
gets()	•	•	•	•
isatty()	•	•	•	•
link()		•	•	
lseek()	•	•	•	•
mkdir()	•	•	•	
mount()	•	•	•	•
open()	•	•	•	•
opendir()	•	•	•	•
perror()	•	•	•	
printf()	•	•	•	
putc()	•	•	•	
putchar()	•	•	•	
puts()	•	•	•	
read()	•	•	•	•
readdir()	•	•	•	•
remove()	•	•	•	
rename()	•	•	•	
rewind()	•	•	•	•
rewinddir()	•	•	•	•
rmdir()	•	•	•	
scanf()	•	•	•	•
setbuf()	•	•	•	•
setvbuf()	•	•	•	•
stat()	•	•	•	•
sync()	•	•	•	
tmpfile()			•	
tmpnam()		•	•	
truncate()	•	•	•	
ungetc()	•	•	•	•
unlink()	•	•	•	
unmount()	•	•	•	•
utime()	•	•	•	
vclean()		•		
vfprintf()	•	•	•	
vprintf()	•	•	•	
vstat()	•	•	•	•
write()	•	•	•	

<u>Key</u>
FFS - TargetFFS-NAND and NOR flash file systems
FAT - TargetFAT DOS-compatible file system
RFS - TargetRFS RAM based file system
ZFS - TargetZFS compressed read-only file system