



Flash Disk Controller (FDC) Description

Copyright Tevero AS

This document may be freely distributed as long as it is not changed without the consent of Tevero AS. The mechanisms described within the document shall not be used in any form of device or application without the consent of Tevero AS.

Table of Contents

1 Introduction.....	4
1.1 Scope.....	4
1.2 Overview.....	4
1.3 Features.....	4
1.4 Definitons.....	6
2 Erase Zone Layout.....	7
2.1 Partition Information.....	7
2.2 Unused Area.....	7
2.3 Block Status Table.....	7
3 Reclaiming Erase Zones.....	8
3.1 Partition Usage Saturation.....	8
4 RAM Usage.....	8
5 Flash Device Fault Handling.....	10
5.1 Flash Device Fault Handling.....	10
5.2 FDC RAM Faults.....	10
6 Application Programming Interfaces (APIs).....	11
6.1 The Flash Driver Application Programming Interface (FD-API).....	11
6.1.1 nnErase.....	12
6.1.2 nnRead.....	13
6.1.3 nnWrite.....	14
6.2 The Flash Protection Application Programming Interface (FP-API).....	15
6.2.1 Pseudo Code Example.....	15
6.2.2 Posix Code Example.....	16
6.2.3 sig_semRead.....	17
6.2.4 sig_semWrite.....	17
6.2.5 wai_semRead.....	17
6.2.6 wai_semWrite.....	18
6.3 The FDC User Application Programming Interface (FU-API).....	19
6.3.1 calcfcs8.....	19
6.3.2 fdcClose.....	19
6.3.3 fdcDelete.....	20
6.3.4 fdcDiagMode.....	20
6.3.5 fdcFormat.....	21
6.3.6 fdcGetBlockSize.....	24
6.3.7 fdcGetEraseValue.....	24
6.3.1 fdcGetInfoString.....	24
6.3.2 fdcGetNumberOfBlocks.....	25
6.3.3 fdcGetNumberOfFreeBlocks.....	25
6.3.4 fdcGetNumberOfObsoleteBlocks.....	25
6.3.5 fdcGetStartOffset.....	26
6.3.6 fdcInit.....	26
6.3.7 fdcInvalidate.....	26
6.3.8 fdcIsUsed.....	27

Flash Disk Controller (FDC) Description

6.3.9 fdcOpen.....	28
6.3.10 fdcRead.....	30
6.3.11 fdcReclaim.....	31
6.3.12 fdcResize.....	32
6.3.13 fdcSetWearlevelLimit.....	33
6.3.14 fdcWrite.....	34
6.3.15 fdcWriteTransparent.....	35
6.3.16 Diagnostic Functions.....	35
<i>Appendix A: FDC Integration Guide.....</i>	36

1 Introduction

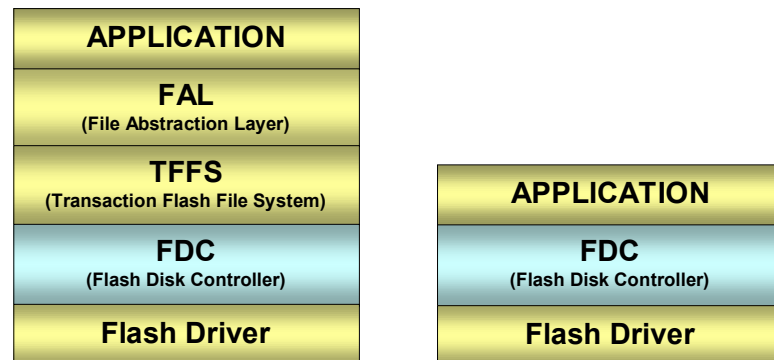
1.1 Scope

The scope of this document is to describe the basic principles and usage of the flash disk controller (FDC). The document does not contain detailed design information, please refer to the source files.

1.2 Overview

The flash disk controller (FDC) is a disk controller used to make a linear flash device array look like a normal disk, hiding the flash related problems with erasing.

FDC is a part of a modular implementation, and can be considered used as the figure below shows:



The leftmost stack shows the usage of FDC in a file system context in order to provide a fault tolerant file system. The rightmost figure shows a simple usage of FDC in order to use it as a block storage device, for instance in order to replace EEPROMs with flash arrays in order to save component costs. (Typically the flash is already present.) Notice that FDC may also be used with other proprietary file systems, however the need for reclaiming requires that the file system is able to mark blocks as obsolete.

1.3 Features

The Flash Disk Controller (FDC) is a system for making a linear flash array look like a disk. FDC provides the following functionality:

- Usable in a preemptive context (e.g. reentrant). This allows for multiple simultaneous read accesses, but only one exclusive write access. (Notice that dual plane devices opens up for reading while writing.) Protection is provided by user defined protection functions per partition as needed. (May not be needed if using similar higher layer protection in for instance a file system.)
- Applicable for uniform linear flash arrays. E.g. a disk partition must consist of uniform sized devices with uniform sized erase zones.
- Supports start-offset that is not a multiple of the partition erase zone size. This implies also that flash devices that hold non-standard sized erase zones can be supported as long as these are not used by FDC.
- Supports holes in flash array usage allowing for reserved boot areas etc.
- Handling an unlimited number of flash disk partitions. A single linear flash array may be split into several individually configured flash partitions.
- Handling of an unlimited number of different vendor linear flash arrays (device driver set up per partition).

Flash Disk Controller (FDC) Description

- Supports removable flash arrays.
- Supports interleaved flash devices (most of the support must be provided by the flash driver)
- The minimum write unit (block) is user defined per partition (minimum 32 bytes). A small block size makes it suitable for long life usage, replacing EEPROMs.
- Each partition may theoretically support up to $2^{28} - 1$ blocks
- Cached bit-chained logical to physical block mapping for fast access. The mapping uses as few bits as possible per mapping entry to reduce RAM usage.
- Wear leveling
- Supports the recovery of any aborted write operation (power failure etc.).
- Supports devices that erase to 1 or to 0.
- Supports dual-plane flash devices (considered as a part of the user provided driver and protection functions).
- Support for NOR types of flash devices (considered as a part of the user provided driver). NAND flash devices are *not* supported. One should ensure that the device supports repeated programming without erase, as long as bits are not changed from used to erased state.
- Handling of bad erase zones and blocks (the max number of bad erase zones/blocks to handle is defined at format time for a partition).
- Resizing a partition (for possible migration from other flash solutions).
- Optional support for 'errno' updating
- Some diagnostics functions supported
- Written in ANSI C.
- Host debugging and development support.

The following is required when using FDC and is not included in FDC:

- Provide read, write and erase function for flash array.
- If used in a preemptive context, protection functions for the selected environment must be implemented if no higher level protection exists.
- Provide the following C library glue functions: *free*, *malloc*, *memcmp*, *memcpy*, *memset*, *strcmp*, *strcpy*, *strlen*, and *printf* (*printf* only if debugging or diagnostics enabled). The functions are defined in *shrdconf.h/fdcconf.h* and they **must** all be reentrant if FDC is used in a preemptive environment.

1.4 Definitions

<i>Block</i>	By a block is meant a contiguous flash memory area holding data typically used by a file system. A file system will normally operate on blocks, binding larger data amounts into a file, which may be scattered around in the flash disk partition. FDC has a way of mapping a logical block into a physical block, so that higher layer SW can modify the same block "indefinitely". Each physical block can only be written once due to status bits used for handling recovery. Once written, it can be made obsolete and reclaimed.
<i>Erase Zone</i>	The minimum area that can be erased in the linear flash array. A flash device consist of erasable blocks or sectors, typically 64KB or 128KB. If the linear flash array consist of interleaved devices (e.g. placed in parallel), then the erase zone represents the number of interleaved devices multiplied with each device's block/sector. (Don't confuse the block term used here with the block term used elsewhere in this document.)
<i>Logical Block</i>	A flash disk partition provides a number of logical blocks to a user. A logical block can be written as many times as liked, it looks like a normal hard disk sector. The main purpose of FDC is to be able to map from logical to physical blocks. Logical blocks are numbered from 0. The user will never know in which physical location a logical block is located, but this does not matter from a user perspective.
<i>Physical Block</i>	The flash disk contains a number of physical blocks for writing/reading. The physical blocks are numbered from 0, and indicate the physical position of its location within the flash array. A physical block can only be written to once. It will have to be made obsolete and then reclaimed, in order to be written to again.
<i>Reclaiming</i>	When a block has been in use and no longer is used, it must be made obsolete. However, an obsolete block can not be reused until the erase zone it resides in is erased. The process of making obsolete blocks reusable, is called reclaiming, and is one of the more complex parts of FDC. Reclaiming has to consider wear leveling, in order to move static blocks around every now and then. FDC will initiate auto-reclaim in any block write operation, if no unused blocks are available. In addition, a system may implement some sort of background reclaiming, in order to provide improved write speed on average.
<i>Wear Leveling</i>	Flash devices typically have a wear limit, e.g. a limit on the number of write operations allowed. If for instance there are 3 erase zones with blocks for data storage, and two of the erase zones holds only static data (e.g. data that never change), that leaves only one erase zone for use for dynamic data. The single erase zone would be worn quite fast. But instead by every now and then move the static data to the erase zone being used the most; we would offload that erase zone. This process is called wear leveling and increases the overall lifetime of the flash array. The penalty is that static data can be moved even if not strictly required.

2 Erase Zone Layout

When a flash partition is formatted with FDC, every erase zone is formatted as follows:

Partition Information
Unused
Block Status Table
Block N
:
Block N + M

The size of the unused field is just padding. Its size depends on the erase zone and block size.

2.1 Partition Information

The partition information holds all information about the flash partition. It is used by *fdcOpen()* to recognize how the partition is formatted and how to use the partition. It also holds information on bad erase zones, the number of times this erase zone has been erased, information needed for recovery etc.

The partition information layout is described in the header of the *fdc.c* source file.

2.2 Unused Area

The unused area is always set as small as possible, in order to make the number of blocks in an erase zone as large as possible.

2.3 Block Status Table

The block status table consist of one entry for every block in that erase zone. The entry size depends on the partition size and may be 1-4 bytes long. Each entry holds status information on the block, and the logical block number for that block. When opening a partition, all valid block status table entries in the complete partition are read to build a logical to physical mapping table.

The block status table entry is described in the header of the *fdc.c* source file.

3 Reclaiming Erase Zones

Eventually the partition goes full due to that all blocks in the partition have been written to. At this point the partition may consist of valid, obsolete and bad blocks. The obsolete blocks can be reclaimed, since we don't need them anymore. In fact, single bad blocks can also be reclaimed, since FDC does not keep track of bad blocks when an erase zone is erased. The block will be remarked as bad when used again, and failing. Notice that bad erase zones will not be reclaimed once marked as bad.

Reclaiming involves selecting an erase zone to reclaim blocks for. Then all valid blocks are moved to unused blocks outside the erase zone being reclaimed, and the erase zone is erased.

Since erasing an erase zone may take a considerable amount of time, frequent and unnecessary reclaiming is not a good idea. However, the write performance of a partition with no free blocks, is poor, since it means that the auto reclaiming is done whenever there are no available unused blocks. However, auto reclaiming will normally free up some additional blocks, unless the disk is very full.

The strategy of when to reclaim is left to the user. It may be done in background mode every now and then when the number of obsolete blocks rises above a certain limit, at start-up or before major write operations. For most systems, the auto-reclaiming done by FDC should be sufficient.

Use `fdcGetNumberOfObsoleteBlocks()` and `fdcGetNumberOfBlocks()` to determine the obsolete rate. The `fdcReclaim()` function can be used to reclaim a certain percentage of obsolete blocks or a certain number of erase zones.

3.1 Partition Usage Saturation

A problem with the flash disk concept arises when the partition fills up. In short, the fuller the partition is, the less number of blocks on average are recovered whenever erasing a flash erase zone. Since erase operations are very time consuming (worst case up to several seconds per erase zone, it depends on the flash device), the partition write performance degrades when the reclaiming frequency increases.

There is no good remedy for this, except dimensioning the partition to be big enough for normal usage.

4 RAM Usage

FDC allocates three memory areas when opening an FDC partition:

- FDC partition data structure
- Erase zone cache table (10-12 bytes for every erase zone)
- Logical to physical mapping table

Of the above memory areas, only the physical mapping table mounts up to any considerable size. The mapping table consists of entries, each entry consuming only the number of bits required to represent all blocks in the partition. Below is a table of this mapping size in bytes for different alternatives.

Flash Disk Controller (FDC) Description

Block size	Erase zone size (KB)	Partition Size (MB)								
		1	2	4	8	16	32	64	128	256
64	64	27356	58620	125056	265744	562752	1188032	2501120	5171712	10835968
	128	27370	58650	125120	265880	563040	1188640	2502400	5177088	10847232
128	64	12974	27944	59880	127744	271456	574848	1213568	2554880	5332992
	128	13000	28000	60000	128000	272000	576000	1216000	2560000	5332992
256	64	6072	13104	28224	60480	129024	274176	580608	1225728	2580480
	128	6084	13130	28280	60600	129280	274720	581760	1228160	2585600
512	64	2794	6096	13208	28448	60960	130048	276352	585216	1235456
	128	2794	6096	13208	28448	60960	130048	276352	585216	1235456
1024	64	1260	2772	6048	13104	28224	60480	129024	274176	580608
	128	1270	2794	6096	13208	28448	60960	130048	276352	585216
2048	64	558	1240	2728	5952	12896	27776	59520	126976	269824
	128	567	1260	2772	6048	13104	28224	60480	129024	274176
4096	64	240	540	1200	2640	5760	12480	26880	57600	122880
	128	248	558	1240	2728	5952	12896	27776	59520	126976

RAM (bytes) needed for mapping for various partition sizes and block sizes

The table above can be used to determine a reasonable block size for a partition. The larger the block size, the less memory is needed for the mapping table, but the more flash memory is wasted if only using part of each block (e.g. lots of short files if FDC is used with a file system).

5 Flash Device Fault Handling

5.1 Flash Device Fault Handling

FDC is to some extent able to handle flash device faults. The reason for such faults would probably be due to exceeding the wear limit. FDC distinguishes between two fault types:

- A fault may occur so that a block can not be written; in which case the block can be marked as bad.
- Status bits in the erase zone can not be written correctly, in which case the complete erase zone must be marked as bad.

This means that faults in the first part of an erase zone are more serious than in the part holding the blocks.

FDC implements a principle of a pool of reclaim blocks. Whenever a bad block is encountered, the reclaim pool will lose one block. Likewise, a bad erase zone will reduce the reclaim pool size by the number of blocks in an erase zone. When the reclaim pool size becomes too small, FDC can no longer handle worst case reclaiming, and the partition will not be writable any more. A worst case reclaiming is the reclaiming of an erase zone with only valid blocks. In this case the same number of reclaim blocks are needed for transferring the valid blocks to. In addition, 2 extra blocks should be available in order to handle the recovery of such a reclaim process.

The user must specify at format time how many blocks shall be reserved for reclaiming. This is done by specifying the number of erase zones (actually the number of blocks each erase zone represents) and an additional number of blocks. Minimum one erase zone and two blocks must be reserved for reclaiming. Some examples of fault handling are:

- Two erase zones and two blocks are reserved. FDC can handle one bad erase zone, but no additional individual block faults. If there are N blocks in an erase zone, FDC can also handle N individual block faults without any erase zone faults.
- Two erase zones and 14 blocks are reserved. FDC can handle one erase zone fault and 12 individual block faults. If there are N blocks in an erase zone, FDC can also handle N+12 individual block faults without any erase zone faults.

How to define the number of reclaim blocks to be reserved is up to the user. In many cases the minimum value would do.

5.2 FDC RAM Faults

Although FDC is made to be able to handle power loss etc., it does use RAM for its internal structures and tables. If the RAM used by FDC is corrupted by other parts of the system, the implications are unpredictable. FDC can not protect itself against such faults without adding considerable overhead.

However, in some cases the consistency between the FDC internal tables and the flash are verified in order to detect such faults. This is done in places where the overhead is considered small compared to the operation to be done. The FDC_FATAL macro in *fdccconf.h* can be defined to invoke a user defined function at such places. The user can then choose to close and reopen the partition or restart the system etc.

6 Application Programming Interfaces (APIs)

6.1 The Flash Driver Application Programming Interface (FD-API)

The flash driver API required by FDC is very simple. It is however extremely important that it is implemented as specified, especially with respect to return values.

One should notice that when using flash devices, the flash device driver themselves is probably the single place where one can set the write access characteristics of the system. Some flash devices have small buffers for parallel writing etc. Use whatever is available to speed up the write access!

The FDC is designed deliberately not to know too much about the flash devices themselves. Support for standards such as the Common Flash Interface (CFI) etc. is considered a driver issue.

One should also notice that it is normally not possible to read from a flash while writing to it. Thus, executing code from flash while trying to use FDC to write on another part is not a straightforward procedure. *(Notice that dual-plane flash devices can be read while being written to: However, optimized support for such devices is up to the user implementation, and not of any relevance to FDC.)*

Please consider the following points when designing the driver. These are just some ideas based on experience, some device types may require other issues to be addressed, or the issues mentioned are not applicable:

1. If using interleaved devices, or devices with buswidth more than 8 bits, make sure that the driver accepts single byte read and write. For instance if using 16 bit mode, the byte before and/or after the data to be written must be read from flash first, and then appended to the data written.
2. Some devices may have a (flash) block/sector locking mechanism. It may be wise to check for locks before writing or erasing (in case it has been accidentally locked). However, some devices may need a considerable amount of time to unlock even if not locked. So the locking mechanism should not be used by the driver unless block/sector actually is locked. Check status bits first instead.
3. If used in a multi-task/process environment (i.e. OS), do consider using interrupts/events when writing/erasing rather than polling for completion. Using interrupts/events in order to wake up the task/process when finished, will allow other tasks/processes with lower priority to be executed while write/erasing is ongoing.
4. If it is possible to write protect the flash device(s) quickly (by for instance defining the address space for the flash as read-only), this may be a good idea. For instance some Intel devices can quite easily be modified by accident by SW using faulty pointers, which may occur, especially in early development phases. The overhead of enabling write access during write can be neglected compared to the rest.
5. It may be wise in the erase function to check if the specified erase zone requires erasing, before actually erasing. This may speed up the first time formatting considerably. The overhead during normal use is none, since FDC will always modify the first byte of an erase zone, and thus the erase function will detect that erase is required immediately during normal use. For instance, if erasing a block takes 1.5 seconds, a 32MB partition with erase zone size 128KB requires 256 erase zones to be erased for a full format. This takes appr. 6.5 minutes! In a manufacturing process where flash usually comes in an erased state, this is unacceptable. The "need for erase" check will then be very useful.

6. It may be wise to ensure that the flash is in normal read mode before doing any operation. It is recommended to ensure that the flash is in normal read mode when finished as well. This overhead can be neglected, and it is a simple safety precaution.
7. The following point is not strictly required by FDC, since FDC will never write or read across a boundary as described below in one operation. It may be wise to consider however if using the driver for non-FDC purposes: Be aware of the special handling that may be required when crossing flash block/sector boundaries or physical flash device boundaries. For instance if writing two bytes to the flash, where the first byte is the last byte in a physical flash device and the second byte is the first byte of another flash device, two devices must be restored to normal read operation. Forgetting to restore the first device, could cause faulty read access by other users, especially if the driver does not restore to normal read mode before doing the operation.
8. Ensure that the flash device actually supports repeated programming to bytes without erase, as long as bits are only changed from erased to used state (normally from 1 to 0). This is normally supported by NOR devices, but Spansion (formerly AMD) has for instance added a limitation feature (ie errata) on some MirrorBit devices. (FDC has a configurable workaround for this particular issue, please see `fdccconf.h`.)

The FD-API functions are defined per FDC partition in the `fdcFormat()` and `fdcOpen()` functions.

If 'errno' support is enabled, this is handled by FDC and higher layers, so the driver should normally not change the 'errno' variable.

6.1.1 nnErase

int nnErase(void *userPointer, U32 offset)

Description: Erase an erase zone. This sets the complete erase zone to 1s or 0s depending on the flash device type in use.

Reentrancy: This function does not have to be reentrant since the user should provide FDC with protection functions if required.

Input: *userPointer* A user pointer, which is, provided transparently in case the `nnErase()` function needs to identify the circumstances for doing the erase. The *userPointer* is provided with the `fdcFormat()` and the `fdcOpen()` functions.

offset The offset to the **first** byte of the erase zone to erase.

Output: *int* 0 indicates that the erase completed.
<0 indicates that the erase did not complete. FDC will treat this as a bad erase zone after trying to erase a second time.

6.1.2 nnRead

int nnRead(void *userPointer, void *to, unsigned int length, U32 offset)

Description: Read bytes from the linear flash array. Notice that the driver must be able to handle the requests of any number of bytes, even if interleaved devices are in use. Single byte reads are quite frequently used by FDC.

FDC does not use any direct memory access into the flash devices, in case the user wants to implement security for ensuring that the flash device are in read mode etc.

Reentrancy: This function must be reentrant since FDC allows multiple reads to occur. Read will never preempt *nnErase()* or *nnWrite()* operations, as long as protection functions for this are provided by the user to FDC.

Input: *userPointer* A user pointer, which is, provided transparently in case the *nnRead()* function needs to identify the circumstances for doing the read. The *userPointer* is provided with the *fdcFormat()* and the *fdcOpen()* functions.

to The pointer to the location to place the bytes read. The buffer must be large enough to hold the specified number of bytes to read.

length The number of bytes to read.

offset The offset to the first byte in the flash array to read.

Output: *int* The **exact** number of bytes read must be returned. <0 indicates that the read could not be done. Notice that FDC treats all return values different from *length* as a fault.

6.1.3 nnWrite

int nnWrite(void *userPointer, const void *buffer, unsigned int length, U32 offset)

Description: Write bytes to the linear flash array. Notice that the driver must be able to handle the requests of any number of bytes, even if interleaved devices are in use. Single byte writes are quite frequently used by FDC.

NOTICE: FDC does not check if the data have actually been written correctly, the driver must provide 100% security for correct writing.

Reentrancy: This function does not have to be reentrant since the user should provide FDC with protection functions if required.

Input: *userPointer* A user pointer, which is, provided transparently in case the *nnWrite()* function needs to identify the circumstances for doing the write. The *userPointer* is provided with the *fdcFormat()* and the *fdcOpen()* functions.

buffer The pointer to the location which holds the bytes to write to the flash.

length The number of bytes to write.

offset The offset to the first byte in the flash array to write.

Output: *int* The exact number of bytes written must be returned. <0 indicates that the write could not be completed successfully. Notice that FDC treats all return values different from *length* as a fault.

6.2 The Flash Protection Application Programming Interface (FP-API)

In order to use in a preemptive system, the user must provide protection functions to FDC if:

- there is no protection on a higher layer (e.g. a file system)
- FDC is going to be used in more than one context

(Notice that the FP-API is NOT required if using TFFS, since it should be provided for TFFS.)

FDC allows only one user to write to a partition at a time. A write operation must complete before any else can read or write. This also implies that a write operation can not start before all read operations are completed.

Notice that dual-plane flashes make it possible to read while writing. However, the user can adapt the protection scheme to whatever flash device is being used, so this should not be a problem. The pseudo code in this chapter does not assume dual-plane flash memory.

FDC allows simultaneous read operations.

In an OS environment, this would most likely mean that one has to use a semaphore, an event and a counter to handle this situation.

If Posix read/write locks are available (`pthread_rwlock_wrlock()` and `pthread_rwlock_rdlock()`), then this is exactly what is needed.

Notice that some devices may support suspend write operations and this can be implemented in the driver and selecting the proper protection algorithms. This approach is not further elaborated here, and would normally not be the first thing to go for.

6.2.1 Pseudo Code Example

An implementation suggestion is indicated below with pseudo-code; be careful to use the defined sequence of operations:

```
U32 readersActive = 0;

int wai_semRead()
{
    /* Wait for possible write to finish. */
    Obtain semaphore

    /* If timeout is supported */
    if (timeout)
        return(-1);

    /* Increment number of readers, this will also inhibit new write */
    /* accesses to take place. We can't rely on semaphore protection here */
    /* since the corresponding releasing read access function does not */
    /* use semaphores. */
    disable preemption
    readersActive++;
    enable preemption

    Release semaphore

    return(0);
}
```

Flash Disk Controller (FDC) Description

```
void sig_semRead()
{
    /* Decrement number of readers. Can't use semaphore here */
    /* since a user waiting for write may have taken it. */
    disable_preemption
    if (readersActive)
        readersActive--;
    enable_preemption

    /* If this was the last one reading, set event in case someone */
    /* is waiting for write access. */
    if (!readersActive)
        set "last read completed" event
}

int wai_semWrite()
{
    /* Get exclusive access to flash, inhibiting new read accesses. */
    Obtain semaphore

    /* If timeout is supported */
    if (timeout)
        return(-1);

    /* Clear read event in case it has been set earlier */
    clear "last read completed" event

    /* If anyone is reading, wait for last reader to finish */
    if (readersActive)
    {
        wait for "last read completed" event

        /* If timeout is supported */
        if (timeout)
        {
            Release semaphore
            return(-1);
        }
    }

    return(0);
}

void sig_semWrite()
{
    /* Let others request access */
    Release semaphore
}
```

6.2.2 Posix Code Example

```
static pthread_rwlock_t rwlock = PTHREAD_RWLOCK_INITIALIZER;

int wai_semRead()
{
    /* Wait for possible write to finish. */
    if (pthread_rwlock_rdlock(&rwlock) < 0)
        return(-1);
}

void sig_semRead()
{
    /* Release lock */
    pthread_rwlock_unlock(&rwlock);
}
```

Flash Disk Controller (FDC) Description

```
int wai_semWrite()
{
    /* Wait for all read operations to finish. */
    if (pthread_rwlock_wrlock(&rwlock) < 0)
        return(-1);
}

void sig_semWrite()
{
    /* Release lock */
    pthread_rwlock_unlock(&rwlock);
}
```

6.2.3 sig_semRead

void sig_semRead(void *userPointer)

Description: End read access to the flash array.

Reentrancy: Reentrant

Input: *userPointer* Pointer provided by user and passed from FDC. Can be used to identify driver type to be used, partition etc.

6.2.4 sig_semWrite

void sig_semWrite(void *userPointer)

Description: End write access to the flash array.

Reentrancy: Reentrant

Input: *userPointer* Pointer provided by user and passed from FDC. Can be used to identify driver type to be used, partition etc.

6.2.5 wai_semRead

int wai_semRead(void *userPointer)

Description: Wait for read access to the flash array. Multiple users may access the flash devices as long as no one is writing to it.

Reentrancy: Reentrant

Input: *userPointer* Pointer provided by user and passed from FDC. Can be used to identify driver type to be used, partition etc.

Output: *int* 0 - access granted
<0 – unable to grant access (FDC will then cancel the operation)

6.2.6 wai_semWrite

int wai_semWrite(void *userPointer)

Description: Wait for write access to the flash array. Only one may write to the flash array at a time, and none may read at the same time.

Reentrancy: Reentrant

Input: *userPointer* Pointer provided by user and passed from FDC. Can be used to identify driver type to be used, partition etc.

Output: *int* 0 - access granted
<0 – unable to grant access (FDC will then cancel the operation)

6.3 The FDC User Application Programming Interface (FU-API)

The user functions are the functions available for user of FDC. A typical user would be a file system, although one may include usage of FDC directly in the application. (One such usage may be to use FDC as a replacement for E²PROMs holding fixed sized parameters.)

6.3.1 calcfcs8

U8 calcfcs8(U8 *data, U8 length)

Description: Calculate 8 bit frame check sequence (FCS) based on $X^8 + X^2 + X + 1$ polynomial. It should detect all odd numbered faults, even numbered faults may not be detected. However, it is better than a plain byte addition checksum. Simulation shows that all single, double and triple bit faults are detected for 15 bytes or less. For 16 to 39 bytes, double faults may not be detected, with the lowest probability of detecting a double bit fault on a 39-byte array of 0.25%.

This function is provided as a service for other usage in a system, since it is used in other contexts than FDC.

Reentrancy: Reentrant

Input: *data* Pointer to first byte to calculate FCS for.

length Number of bytes to calculate FCS for.

Output: *U8* Calculated frame check sequence

6.3.2 fdcClose

int fdcClose(int id)

Description: Close a flash partition. All associated resources are freed.

Reentrancy: Reentrant (if protection functions provided with *fdcOpen()*)

Input: *id* Partition ID for partition to close. This id was obtained with the *fdcOpen()* function.

Output: *int* 0 - partition closed successfully
<0 - trying to close non-opened partition

If 'errno' usage is enabled, 'errno' is updated on error to: ENXIO or EBUSY

6.3.3 fdcDelete

int fdcDelete(int id, U32 block)

Description: Delete a logical block, e.g. mark it as obsolete. It is important that a user deletes flash blocks which are not needed anymore. If not, then the whole partition may become full, requiring an erase zone to be erased for every block written. This will lead to a disastrous write performance, since erase zones must be reclaimed for every write!

Notice that this function does not start any reclaiming process, it just marks the block as obsolete.

Reentrancy: Reentrant (if protection functions provided with *fdcOpen()*)

Input:

<i>id</i>	Partition ID for partition to delete block
<i>block</i>	Logical block number to delete (make obsolete)

Output:

<i>int</i>	0 - block deleted <0 - some fault occurred
------------	---

If 'errno' usage is enabled, 'errno' is updated on error to: ENXIO, EINVAL, EIO or EBUSY

6.3.4 fdcDiagMode

int fdcDiagMode(int id, U8 mode)

Description: Control diagnostic mode for a partition.

This function is only included if diagnostic is enabled in *fdcconf.h* (FDC_DIAG set to YES)

Reentrancy: Reentrant

Input:

<i>id</i>	Partition ID for partition to set diagnostic mode
<i>mode</i>	Diagnostic mode to set. Use one of: FDC_DIAG_DISABLE - Disable diagnostics logging FDC_DIAG_ERRORS - Show errors FDC_DIAG_ALL - Show all diagnostics actions and errors

Output:

<i>int</i>	Returns previous diagnostic mode <0 - Invalid partition ID and no change done.
------------	---

If 'errno' usage is enabled, 'errno' is updated on error to: ENXIO, EINVAL or EBUSY

6.3.5 fdcFormat

```
int fdcFormat(
    char *info,
    U32 startOffset,
    U32 partitionSize,
    U32 partitionMaxSize,
    U8 deviceSize,
    U8 ezoneSize,
    U8 blockSize,
    U8 layout,
    U8 reclaimEzones,
    U8 reclaimBlocks,
    void *userPointer,
    int (*flashErase)(void *userPointer, U32 offset),
    int (*flashRead)(void *userPointer, void *to, unsigned int length, U32 offset),
    int (*flashWrite)(void *userPointer, const void *buffer, unsigned int length, U32 offset),
    int (*wai_semWrite)(void *userPointer),
    void (*sig_semWrite)(void *userPointer),
    U8 diagMode )
```

Description: Format a linear flash partition in order to be able to use with the FDC system. All data in the defined partition will be lost. When completed, the partition may be opened.

If the format is aborted, an opening may succeed if the partition information header in at least one erase zone is completed. Opening will then complete the formatting.

NOTICE:

This function allocates memory needed for various issues. The memory is kept only during formatting.

Reentrancy: Reentrant on different non-overlapping partitions.

Input:	<i>info</i>	Pointer to 0 terminated info string (user definable). Must not be longer than FDC_MAX_INFO characters. The string is included in the partition info and can be used for identification.
	<i>startOffset</i>	Offset from start of flash devices (as seen by the driver) that the partition starts. The offset must be pointing to the beginning of an erase zone.
	<i>partitionSize</i>	The number of erase zones to be used by the partition.
	<i>partitionMaxSize</i>	If the partition shall be expanded in the future, this must be planned at format time in order to reserve enough space for entries in the Block Status Table. If set to a value less than <i>partitionSize</i> , it will automatically be set to <i>partitionSize</i> , and the partition can not be expanded in the future.
	<i>deviceSize</i>	The size of physical flash device that make up the linear flash array. The size is N in 2^N .

Flash Disk Controller (FDC) Description

<i>erzoneSize</i>	The size of an erase zone which is the minimum that can be erased. The size is N in 2^N . Notice that if physical flash devices are interleaved to form flash array with greater bitwidth, the erase zone size shall reflect the total erase zone size of all the interleaved devices added together.
<i>blockSize</i>	The block size is the operating unit that a higher layer system can read and write to. It is N in 2^N and can not be less than 5 (e.g. minimum blocksize is 32).
<i>layout</i>	Defines the layout of the flash array. The 4 MSB defines the interleaving, e.g. number of devices connected in parallel. The 4 LSB defines the bit width of a single device alone. Use the FDC_DEVICES_INTERLEAVED_x and FDC_DEVICE_BITWIDTH_x defines ORed together.
<i>reclaimEzones</i>	Number of erase zones to reserve for reclaiming. Use N in N + 1, to indicate the number of erase zones to be reserved. More than 0 is only required if acceptance of faulty erase zones shall be handled. Notice that both bad erase zones and blocks will eat from the reserved blocks defined by reclaimEzones and reclaimBlocks. When fewer blocks than in one erase zone plus one are usable, the partition will not be writable anymore.
<i>reclaimBlocks</i>	Number of blocks to reserve for reclaiming. Use N in N + 1, to indicate the number of blocks. Notice that 0 is OK, but in some cases recovery can not be handled with only one reserved block, so 1 (=2 blocks) is minimum if reclaimEzones is 0.
<i>userPointer</i>	Pointer that can be set by the user and possibly used by user provided functions that it is passed on to. Not used by FDC.
<i>flashErase</i>	Function pointer to device driver used to erase erase zones in the flash array.
<i>flashRead</i>	Function pointer to device driver used to read from the flash array.
<i>flashWrite</i>	Function pointer to device driver used to write to the flash array.
<i>wai_semWrite</i>	Function pointer to function waiting for write access semaphore for the flash device. Set to NULL if the flash access shall be treated without preemption concerns.
<i>sig_semWrite</i>	Function pointer to function releasing the write access semaphore for the flash device. Set to NULL if the flash access shall be treated without preemption concerns.
<i>diagMode</i>	Diagnostic mode to use while formatting. Use FDC_DIAG_xxx defines. Only used if FDC_DIAG is enabled in fdccconf.h
Output:	
<i>int</i>	0 - format completed successfully <0 - error during formatting

If 'errno' usage is enabled, 'errno' is updated on error to: EINVAL, ENOMEM, EEXIST, EIO or EBUSY

6.3.6 fdcGetBlockSize

int fdcGetBlockSize(int id, U8 *size)

Description: Get block size, N in 2^N

Reentrancy: Reentrant

Input: *id* Partition ID for partition to get block size

number Location to store fetched block size

Output: *int* 0 - Block size fetched
<0 - Fault, number is not fetched.

If 'errno' usage is enabled, 'errno' is updated on error to: ENXIO

6.3.7 fdcGetEraseValue

int fdcGetEraseValue(int id)

Description: Get what the flash erases to, 0 or 1. This function is provided for very limited use, mostly if transparent write is required.

Reentrancy: Reentrant

Input: *id* Partition ID for partition to get erase value

Output: *int* 0 - Flash erases to 0
1 - Flash erases to 1
<0 - Fault, value is not fetched.

If 'errno' usage is enabled, 'errno' is updated on error to: ENXIO

6.3.1 fdcGetInfoString

const char *fdcGetInfoString(int id)

Description: Get pointer to flash disk partition info string. The partition info string is defined at format time, and can be used to identify the partition.

Reentrancy: Reentrant

Input: *id* Partition ID for partition to get info string pointer.

Output: *const char ** Ptr to partition info string. Notice that the string can be 0 length. NULL is returned if invalid ID is provided. The string is statically located and must not be modified.

6.3.2 fdcGetNumberOfBlocks

int fdcGetNumberOfBlocks(int id, U32 *number)

Description: Get number of blocks in partition. This is the number of usable blocks, denoted logical blocks internally in FDC.

Reentrancy: Reentrant

Input: *id* Partition ID for partition to get number of blocks
number Pointer to location to place number of blocks

Output: *int* 0 - Number of blocks in partition fetched
<0 - Fault, number is not fetched

If 'errno' usage is enabled, 'errno' is updated on error to: ENXIO

6.3.3 fdcGetNumberOfFreeBlocks

int fdcGetNumberOfFreeBlocks(int id, U32 *number)

Description: Get number of free (unused) blocks in partition. Can typically be used together with fdcGetNumberOfObsoleteBlocks() and fdcGetNumberOfBlocks() to determine how much of the partition that is used.

Reentrancy: Reentrant

Input: *id* Partition ID for partition to get number of free blocks.
number Pointer to location to place number of free blocks

Output: *int* 0 - Number of free blocks in partition fetched
<0 - Fault, number is not fetched.

If 'errno' usage is enabled, 'errno' is updated on error to: ENXIO

6.3.4 fdcGetNumberOfObsoleteBlocks

int fdcGetNumberOfObsoleteBlocks(int id, U32 *number)

Description: Get number of obsolete blocks in partition. Can typically be used to determine when to start reclaiming processes.

Reentrancy: Reentrant

Input: *id* Partition ID for partition to get number of obsolete blocks.
number Pointer to location to place number of obsolete blocks

Output: *int* 0 - Number of obsolete blocks in partition fetched
<0 - Fault, number is not fetched.

If 'errno' usage is enabled, 'errno' is updated on error to: ENXIO

6.3.5 fdcGetStartOffset

int fdcGetStartOffset(int id, U32 *offset)

Description: Get start offset in flash device for a partition as defined when formatting the partition.

Reentrancy: Reentrant

Input: *id* Partition ID for partition to get start offset.
offset Pointer to location to place start offset.

Output: *int* 0 - Start offset for partition fetched
<0 - Fault, start offset is not fetched

If 'errno' usage is enabled, 'errno' is updated on error to: ENXIO

6.3.6 fdclnit

void fdclnit(U32 max)

Description: Initialize FDC system. Should be invoked once before using FDC.

Reentrancy: Reentrant

Input: *max* Max number of FDC partitions that can be defined. A pointer is allocated for every partition that can be defined. The pointer table is later used for fast lookup.

6.3.7 fdclinvalidate

int fdclinvalidate(int id)

Description: Invalidate and close an opened flash partition. This function will invalidate all erase zone by simply just invalidating the header of each erase zone. It does not erase the erase zones, and thus is fairly fast.

It may be used for instance if reformatting a partition to a different size, and to ensure no usable traces are left of previous partition.

Reentrancy: Reentrant

Input: *id* Partition ID for partition to invalidate and close (the partition must be opened.)

Output: *int* 0 - Flash partition invalidated and closed
<0 - Fault, flash partition invalidation not completed

If 'errno' usage is enabled, 'errno' is updated on error to: ENXIO, EIO, EBUSY

6.3.8 fdclsUsed

int fdclsUsed(int id, U32 block)

Description: Check if a block is used (e.g written to) or not.

Reentrancy: Reentrant

Input: *id* Partition ID for partition to check if used block

block Logical block to check if used

Output: *int* 1 - Block is used (e.g. contains data)
 0 - Block is not used (and does not contain valid data).
 <0 - Illegal usage (wrong parameters)

If 'errno' usage is enabled, 'errno' is updated on error to: ENXIO, EINVAL or EIO

6.3.9 fdcOpen

```
int fdcOpen(
    U32 startOffset,
    U32 range,
    U8 step,
    void *userPointer,
    int (*flashErase)(void *userPointer, U32 offset),
    int (*flashRead)(void *userPointer, void *to, unsigned int length, U32 offset),
    int (*flashWrite)(void *userPointer, const void *buffer, unsigned int length, U32
offset),
    int (*wai_semRead)(void *userPointer),
    void (*sig_semRead)(void *userPointer),
    int (*wai_semWrite)(void *userPointer),
    void (*sig_semWrite)(void *userPointer),
    U8 diagMode )
```

Description: Open a flash disk partition. The partition is scanned and the following are done as part of the scanning:

- recover any abnormal previous termination:
- build logical to physical block mapping table
- build erase zone cache table
- build partition info status

NOTICE:

This function allocates memory needed for various issues. The memory is kept until closing the partition.

Reentrancy: Reentrant on different non-overlapping partitions.

Input: *startOffset* Offset in bytes to start looking for partition. This offset is just passed onto the flash driver. The offset will automatically be aligned to 1KB aligned offset.

Usually the exact offset is known, then use that offset.

range Number of bytes to scan for partition. If 0, then 1MB is scanned.

NOTICE: It is important to specify a range that will cover at least 2 erase zones. If only specifying looking within one erase zone, and that erase zone was being reclaimed when the system was aborted, the partition may not be found when opening since the erase zone may be in an erased state.

step N in 2^N indicating the step from startOffset to increment for every failing trial. If set to 0, the minimal step size (1KB) is selected. Typically the step size can be set equal to the erase zone size if the erase zone size is known.

userPointer Pointer that can be set by the user and possibly used by user provided functions that it is passed on to. Not used by FDC.

flashErase Function pointer to device driver used to erase erase zones in the flash array.

flashRead Function pointer to device driver used to read from the flash array.

flashWrite Function pointer to device driver used to write to the flash array.

Flash Disk Controller (FDC) Description

<i>wai_semRead</i>	Function pointer to function waiting for read access semaphore for the flash device. Set to NULL if the flash access shall be treated without preemption concerns.
<i>sig_semRead</i>	Function pointer to function releasing the read access semaphore for the flash device. Set to NULL if the flash access shall be treated without preemption concerns.
<i>wai_semWrite</i>	Function pointer to function waiting for write access semaphore for the flash device. Set to NULL if the flash access shall be treated without preemption concerns.
<i>sig_semWrite</i>	Function pointer to function releasing the write access semaphore for the flash device. Set to NULL if the flash access shall be treated without preemption concerns.
<i>diagMode</i>	Diagnostic mode to use for partition. Use FDC_DIAG_XXX defines. Only used if FDC_DIAG is enabled.

Output: *int* >=0 - The ID of the partition opened
 <0 - unable to open partition

If 'errno' usage is enabled, 'errno' is updated on error to: ENOMEM, EINVAL, EBUSY, EEXIST, ENOSPC or EIO

6.3.10 fdcRead

int fdcRead(int id, U32 block, U8 *to, U32 length, U32 offset)

Description: Read data from a block in a flash disk partition. This provides the user access point to FDC, where a logical block is mapped into a physical block on the flash disk and then read.

Reentrancy: Reentrant (protection functions must be provided as needed) There may be multiple simultaneous reads as long as no one is writing.

Input:

<i>id</i>	Partition ID for partition to read block from
<i>block</i>	Logical block to read data from
<i>to</i>	Pointer to location to place read data into
<i>length</i>	Number of bytes to read from block. Should not be larger than the size of a block.
<i>offset</i>	Offset from start of block to read bytes from.

Output: *int* 0 if requested data read. If the logical block does not exist (never written to or made obsolete), this is considered a fault by FDC.
 <0 if some fault occurred.

If 'errno' usage is enabled, 'errno' is updated on error to: ENXIO, EINVAL, EIO or EBUSY

6.3.11 fdcReclaim

int fdcReclaim(int id, U8 percent)

Description: Reclaim obsolete blocks in a partition up to a certain percentage or just one erase zone.

Notice that even if write protection is asserted (if the FDC partition has been defined with a protection scheme) for every erase zone being reclaimed, it is de-asserted between every erase zone being reclaimed. This allows the reclaim operation to be interrupted by other partition accesses.

Reentrancy: Reentrant (protection functions must be provided as needed)

Input:

<i>id</i>	Partition ID for partition to do reclaim
<i>level</i>	If MSB is 0: Percent of obsolete blocks to reclaim (0-100). This means that if there are 100 obsolete blocks, and this parameter is 70, the reclaiming will stop once at least 70 blocks have been reclaimed.

0x80:
Reclaim just one erase zone (not done if no obsolete blocks exist). **Notice** that it is not guaranteed that any obsolete blocks are reclaimed, since every now and then erase zones may be "reclaimed" due to wear leveling, even though they do not have any obsolete blocks.

Output:

<i>int</i>	0 - reclaiming completed <0 - some fault occurred
------------	--

If 'errno' usage is enabled, 'errno' is updated on error to: ENXIO, EINVAL, EIO, ENOSPC or EBUSY

6.3.12 fdcResize

int fdcResize(int *id, U16 ezones, U8 front, U8 expand)

Description: Used to resize a partition. When the resizing is completed, the partition is automatically closed and reopened. Thus, the partition ID may change when resizing.

(This function has been implemented mainly for supporting transitions to using FDC. E.g. at startup part of a flash array may consist of some data. A partition is created in an unused part of the flash array, the data is moved into the FDC partition, and then the partition is expanded to cover the earlier used area as well.)

NOTE:

Only expansion is currently supported.

Resizing is currently not recoverable! In order to make it recoverable, some extra functionality must be added to fdcOpen().

NOTICE:

This function allocates memory needed for various issues. The memory is kept only during resizing.

Reentrancy: Reentrant (protection functions must be provided as needed)

Input: *id* Pointer to partition ID for partition to resize. If the resize completes, then the partition is closed and reopened, thus the partition ID may change.

ezones Number of erase zones to expand/reduce, max $2^{14} - 1$

front 0 - Resize at end of current partition
1 - Resize at the beginning of current partition

expand 0 - Reduce (currently not supported)
1 - Expand

Output: *int* 0 - Resizing completed
<0 - Unable to resize

If 'errno' usage is enabled, 'errno' is updated on error to: ENXIO, EINVAL, ENOMEM, ENOSPC, EBUSY or EIO

6.3.13 fdcSetWearlevelLimit

U32 fdcSetWearlevelLimit(int id, U32 limit)

Description: Set wear level limit for a partition.

If an erase zone is filled with static data, then other erase zones will be used more than those holding the static data. If an erase zone has been erased less than 'limit' times less than the otherwise selected erase zone, it will be reclaimed to move the data to an erase zone erased more often.

The default wear level limit is 300.

It should normally not be required to use this function.

Reentrancy: Reentrant

Input:	<i>id</i>	Partition ID for partition to set limit for
	<i>limit</i>	Wear level limit to set for partition (≥ 100)
Output:	<i>U32</i>	Old wear level limit (0 if invalid partition)

6.3.14 fdcWrite

int fdcWrite(int id, U32 block, const U8 *from, U32 length)

Description: Write data to a block on a flash disk. This provides the user access point to FDC, where a logical block is mapped into a physical block on the flash disk and then written to.

Notice that although less than the whole block may be written (always starting from offset 0 in the block), data can not later be appended to the unused part of the block. This follows from that bits used for recovery purposes have been used, and can not be reused due to general flash behavior.

This implies that a file system should cache minimum a block before writing it to the flash media.

Reentrancy: Reentrant (protection functions must be provided as needed). Only one may be writing at a time, and no read operation is allowed during this.

Input:

<i>id</i>	Partition ID for partition to do write
<i>block</i>	Logical block number to write data to.
<i>from</i>	Pointer to location to place write data from
<i>length</i>	Number of bytes to write to block. Must not be larger than the available number of bytes in a block.
<i>category</i>	Used to indicate what type of data is being written. Data of a certain category will be placed as far as possible in erase zones of the same category. This can be used by higher layer SW to increase reclaiming efficiency. Supported categories are: FDC_CATEGORY0 and FDC_CATEGORY1.

Output:

<i>int</i>	0 if write successful. <0 if some fault occurred
------------	---

If 'errno' usage is enabled, 'errno' is updated on error to: ENXIO, EINVAL, ENOSPC, EIO or EBUSY

6.3.15 fdcWriteTransparent

int fdcWriteTransparent(int id, U32 block, const U8 *from, U32 length, U32 offset)

Description: Write data to a block on a flash disk that may already have been written to. This write operates in transparent mode, not modifying the status bits for the block (except for the first time writing to an unused block). Thus, the write is not recoverable except the first time writing to a block.

WARNING: Use this function with care!

Reentrancy: Reentrant (protection functions must be provided as needed) Only one may be writing at a time, and no read operation is allowed during this.

Input:

<i>id</i>	Partition ID for partition to do write
<i>block</i>	Logical block number to write data to.
<i>from</i>	Pointer to location to place write data from
<i>length</i>	Number of bytes to write to block. Must not be larger than the available number of bytes in a block.
<i>offset</i>	Offset in block to start writing to. This value must be 0 the first time writing to a block (e.g. writing to an unused block).
<i>category</i>	Used to indicate what type of data is being written. Data of a certain category will be placed as far as possible in erase zones of the same category. This can be used by higher layer SW to increase reclaiming efficiency. Supported categories are: FDC_CATEGORY0 and FDC_CATEGORY1.

Output: *int* 0 if write successful.
<0 if some fault occurred

If 'errno' usage is enabled, 'errno' is updated on error to: ENXIO, EINVAL, EBUSY or EIO

6.3.16 Diagnostic Functions

Some diagnostic functions are provided, and more may be provided in the future. Please refer to the source code for available functions. (Diagnostics functions are located in *fdcd*.c* files.)

The diagnostics functions use the *fdcPrintf* macro defined in *fdconf.h*, which normally in turn refers to *shrdPrintf* macro defined in *shrdconf.h*. This function must be provided in order to make any use of the diagnostics functions.

Appendix A: FDC Integration Guide

This guide assumes that you have a basic knowledge of your compiler and build environment. It provides a step-by-step instruction in order to start using FDC on your target.

1. **Configuration:** Do the entire configuration required by modifying the *src/fdc/fdcconf.h* and *src/shared/shrdconf.h* files. It is recommended that debugging is enabled at first, if some support for *printf()* is provided.
2. **Ensure that FDC compiles in your environment:** First include the FDC source files (*src/fdc/*.c* and *src/shared/*.c*) in your make or build environment. You should ensure that all the include files (*src/fdc/*.h* and *src/shared/*.h*) are available when compiling the source files.
3. **Flash driver:** Design and **test** the flash device driver to be used for the target. Please refer to “The Flash Driver Application Programming Interface (FD-API)” chapter for details.
4. **Protection:** If FDC is to be used without higher level reentrancy protection in a multi-user environment, design the protection functions as described in the “The Flash Protection Application Programming Interface (FP-API)” chapter.
5. **Opening and formatting:** Below is a code example on how to open an 8MB flash partition. The flash partition consist of 2 4MB devices with 64KB block/sectors (erase zones) designed in serial (thus no interleaving).

```
int applFdcOpen()
{
    int ret;

    /* Must be done once before any use of FDC, there will only be 1 partition */
    fdcInit(1);

    /* Try to open FDC partition, format if not able to open it */
    ret = fdcOpen(0,          /* Start looking from the beginning
                             of the first flash */
                 0,          /* Expect to find the partition
                             within the first 1MB */
                 0x10000,    /* Use 64KB stepping when searching */
                 NULL,
                 flashErase,
                 flashRead,
                 flashWrite,
                 NULL,        /* No reentrancy protection provided */
                 NULL,
                 NULL,
                 NULL,
                 FDC_DIAG_ALL);

    if (ret < 0)
    {
        ret = fdcFormat("EXAMPLE",
                       0,          /* Disk starts at the beginning */
                       0x800000 / 0x10000, /* Number of erase zones */
                       0,          /* We don't plan to expand the
                                   partition in the future */
                       22,        /* 4MB (2^22) devices */
                       16,        /* 64KB (2^16) erase zone size */
                       9,         /* Block size 512 (2^9) bytes */
                       FDC_DEVICES_INTERLEAVED_1 | FDC_DEVICE_BITWIDTH_8,
                       0,          /* 1 ezone reserved for reclaiming */
                       2,         /* Additional 3 blocks reserved
                                   for reclaiming */
                       NULL,
                       flashErase,
                       flashRead,
                       flashWrite,
                       NULL,        /* No reentrancy protection provided */
                       NULL,
    }
```

Flash Disk Controller (FDC) Description

```
        FDC_DIAG_ALL);
    if (ret < 0)
    {
        printf("FDC format failed\n");
        return(ret);
    }

    /* Same open as above */
    ret = fdcOpen(0,
                 0,
                 0x10000,
                 NULL,
                 flashErase,
                 flashRead,
                 flashWrite,
                 NULL,
                 NULL,
                 NULL,
                 NULL,
                 FDC_DIAG_ALL);

    if (ret < 0)
    {
        printf("FDC open failed\n");
        return(ret);
    }
}

return(ret);
}
```

And the application would look something like below:

```
int fdcHandle; /* Keep this for later use */

fdcHandle = applFdcOpen(...);
if (fdcHandle < 0)
{
    /* Can't use FDC for some reason.... */
}
```

If all this has succeeded, the next step is to start using FDC by writing and reading blocks. (Notice that the FDC mechanisms of reclaiming must be understood in order to use FDC standalone.)